Co-funded by the
Erasmus+ Programme
of the European Union

Building Distributed Systems for High-Stress Environments using Reversibility and Phase-Awareness

RUMA R. PAUL

Doctoral Thesis in Information and Communication Technology
School of Information and Communication Technology
KTH Royal Institute of Technology
Stockholm, Sweden 2016
and
Institute of Information and Communication Technologies,
Electronics and Applied Mathematics
Université catholique de Louvain
Louvain-la-Neuve, Belgium 2016

KTH School of Information and
Communication Technology
SE-164 40 Kista
SWEDEN

TRITA-ICT 2016:26
ISBN 978-91-7729-112-1

Akademisk avhandling som med tillstånd av Kungl Tekniska högskolan framlägges till offentlig granskning för avläggande av doktorsexamen i Informations- och Kommunikationsteknik tisdag den 11 oktober 2016 klockan 13:00 i Sal B, Electrum, Kungl Tekniska högskolan, Kistagången 16, Kista.

Tryck: Universitetsservice US AB

Abstract

A paradigm shift is underway in the field of distributed computing systems. From data-center based computing, we are moving towards edge computing, i.e., pushing the computation towards the logical extreme of a network. Thus, the operating environment of distributed systems continues to become more stressful due to inherent unreliability and heterogeneity of the edge networks. For example, in case of mobile and ad hoc networks *Churn* (node turnover) can be extremely high due to node mobility, frequent disconnects/reconnects and configuration changes. Also, in such dynamic environments, the system may face high churn and network partitioning frequently. In this thesis, we propose a generic approach for building distributed applications that lets them survive, but still providing useful service to the users, in high-stress operating environments such as these.

To survive in stressful environments such as edge networks, the system needs to have strong self-healing properties. We make empirical studies of both the behavior and design of a system in high-stress environments. In other terms, we seek answers to the following non-trivial questions: Can a system heal the damage due to stress in its operating environment and regain its full functionality as the stress fades away? Can we predict the behavior of the system, i.e., available functionality during the stress? What are the preconditions; in other words, what properties should the healing mechanism possess in order to achieve these objectives against stressful environments?

In order to answer these questions we first introduce and formalize the concepts of *Reversibility* and *Phase*. Reversibility implies the functionality of a system (the set of services that it can provide satisfactorily) is a function (the *reversibility function*) of its current stress level in its operating environment and does not depend on the history of the stress. As stress is a global condition that cannot easily be measured by individual nodes, we propose the concept of *Phase* in order to approximate the reversibility function. Phase is a per-node property that gives a qualitative indication of the system functionality available at that node. Phase can be determined with no additional distributed computation. In this thesis we explain the usefulness of these concepts by means of two case studies.

We build on the concept of Structured Overlay Network (SON), a well known approach to building decentralized distributed systems. In our *First Case Study* we extend an existing SON that hosts a transactional replicated key-value store to be *Reversible*, which implies the system is able to regain its original functionality as the stress (e.g., churn, network partitioning, physical network dynamicity etc.) declines. We evaluate reversibility of the system using existing overlay maintenance strategies, namely Correction-on-Change, Correction-on-Use, Periodic Stabilization, and Ring Merge. We propose a new resilient maintenance mechanism, called *Knowledge Base*, to improve conditions for reversibility against inhospitable environments. We identify the necessary maintenance strategies to achieve reversibility. By means of simulations, we demonstrate reversibility for an overlay network with high levels of partition, churn and packet-loss. We also analyze the interaction between two stress elements, namely churn and network partitioning, and identify the limits of simultaneous partitioning and churn, up to which the system can achieve reversibility by itself. We make general conclusions about the ability of the maintenance strategies to achieve reversibility.

In order to approximate the reversibility function, and thus describe the qualitative behavior of a system, we introduce the concept of *Phase*. All nodes in a phase exhibit the same qualitative properties, which are different for nodes in different phases. As the phase at each node provides a qualitative indication of what system operations are available, it allows the applications, running on top of the system, to manage their behavior predictably in stressful environments. We demonstrate the existence of *Reversible Phase Transitions* (i.e., a significant fraction of nodes that change phase) in the system while experiencing stress and show that our concept of phase is analogous to the macroscopic phase of physical systems. We empirically identify the Critical Points (i.e., when there exists more than one phase simultaneously in significant fractions of the system) observed in our experiments. We extend each node of the system to be self-aware to compute its phase without any global synchronization. We propose an API to allow the application layer to

be informed about the current phase of a node. In our *Second Case Study* we present a use-case application, a distributed collaborative graphic editor, built on top of the key-value store, and we explain how to make it Phase-Aware, i.e., how to make it optimize its behavior according to a real-time observation of phase at each node. Furthermore, we analyze how the application itself can achieve reversibility for the application-level semantics. Using the phase of the underlying node, the application provides indication to the user regarding its behavior (both current and expected). Thus, the application has improved behavior with respect to the user, i.e., the user can understand and decide what to do in a high-stress environment.

Our case studies show the usefulness of reversibility for building distributed systems and of phase-awareness for building applications. Reversibility gives improved functionality (internal to the application) and phase-awareness gives improved usability (external user). Thus, based on these concepts, in this thesis we prepare the way to build provably correct complex computing systems, particularly distributed systems and applications having improved usability, for arbitrarily stressful operating environments.

Résumé

Un changement de paradigme est en cours dans le domaine des systèmes informatiques distribués. Á la place du calcul en centre de données, nous avançons vers le calcul fait en réseau de bord. En conséquence, l'environnement opérationnel des systèmes distribués continue à devenir plus stressant en raison du manque de fiabilité inhérente et de l'hétérogénéité des réseaux de bord. Par exemple, dans le cas de la téléphonie mobile et les réseaux ad hoc, *l'Attrition* (la proportion des nœuds ayant changé par unité de temps) peut être extrêmement élevée en raison de la mobilité des nœuds, des fréquentes déconnexions / reconnexions et des changements de configuration. En outre, dans de tels environnements dynamiques, le système peut faire face à un haut taux d'attrition et du partitionnement du réseau d'une manière fréquente. Dans cette thèse, nous proposons une approche générique pour la construction d'applications distribuées qui leur permet de survivre d'environnements opérationnels de haut stress tels que ceux-ci, en fournissant toujours un service utile pour les utilisateurs.

Pour survivre dans des environnements stressants tels que les réseaux de bord, le système a besoin de fortes propriétés d'auto-guérison. Nous faisons des études empiriques à la fois du comportement et de la conception d'un système dans un environnement à stress élevé. En d'autres termes, nous cherchons des réponses aux questions non triviales suivantes : Un système peut-il guérir les dommages dus au stress dans son environnement opérationnel et retrouver sa pleine fonctionnalité quand le stress disparaît ? Peut-on prédire le comportement du système, à savoir, la fonctionnalité disponible pendant le stress ? Quels sont les conditions préalables ; en d'autres termes, quelles propriétés devraient posséder le mécanisme de guérison pour atteindre ces objectifs dans un environnement stressant ?

Afin de répondre à ces questions, nous présentons et formalisons d'abord les concepts de *Réversibilité* et de *Phase*. La Réversibilité implique que la fonctionnalité d'un système (l'ensemble de services qu'il peut fournir de façon satisfaisante) est une fonction (appellée la fonction de réversibilité) de son niveau de stress actuel dans son environnement opérationnel et ne dépend pas de l'histoire de ce niveau. Comme le stress est une condition globale qui ne peut pas être facilement mesuré par des nœuds individuels, nous proposons le concept de *Phase* afin d'approcher la fonction de réversibilité. La Phase est une propriété de chaque nœud qui donne une indication qualitative de la fonctionnalité du système disponible à ce nœud. La Phase peut être déterminée sans aucun calcul distribué supplémentaire. Dans cette thèse, nous expliquons l'utilité de ces concepts au moyen de deux études de cas.

Nous nous appuyons sur le concept de réseau superposé structuré (Structured Overlay Network - SON), une approche bien connue pour la construction de systèmes distribués décentralisés. Dans notre *Premier Étude de Cas*, nous étendons un SON existant, qui réalise un système de stockage clé-valeur transactionnel répliqué, pour être réversible, ce qui implique que le système est en mesure de retrouver sa fonctionnalité originale quand le stress (par exemple, l'attrition, le partitionnement de réseau, la dynamicité du réseau physique, etc.) diminue. Nous évaluons la réversibilité du système en utilisant des stratégies de maintenance existantes, à savoir Correction-on-Change, Correction-on-Use, la stabilisation périodique, et la fusion d'anneau. Nous proposons un nouveau mécanisme de maintenance, appelé *Base de Connaissances*, pour améliorer les conditions de réversibilité contre les environnements inhospitaliers. Nous identifions les stratégies de maintenance nécessaires pour atteindre la réversibilité. Au moyen de simulations, nous démontrons la réversibilité pour un réseau superposé avec des niveaux élevés de partitionnement, attrition et taux de perte de paquets. Nous analysons également l'interaction entre deux paramètres de stress, à savoir l'attrition et le partitionnement, et nous identifions les limites de l'attrition et du partitionnement simultanés, jusqu'Ã laquelle le système peut atteindre la réversibilité par lui-même. Nous faisons des conclusions générales quant à la capacité des stratégies de maintenance pour atteindre la réversibilité.

Afin d'approcher la fonction de réversibilité, et donc de décrire le comportement qualitatif d'un système, nous introduisons le concept de Phase. Tous les nœuds dans la même phase présentent les mêmes propriétés qualitatives, qui sont différentes pour les nœuds d'autres phases. Comme la phase, à chaque

nœud , fournit une indication qualitative des opérations disponibles, il permet aux applications en cours d'exécution sur le système, de gérer leur comportement de façon prévisible dans un environnement stressant. Nous démontrons l'existence de *Transitions de Phase Réversibles* (à savoir, une fraction importante des nœuds qui changent de phase) dans un système soumis au stress et nous montrons que notre concept de phase est analogue à la phase macroscopique des systèmes physiques. Nous identifions empiriquement les *Points Critiques* (c'est-à-dire, lorsqu'il existe plus d'une phase simultanément dans des fractions importantes du système) observés dans nos expériences. Nous étendons chaque nœud du systmè à calculer sa phase sans aucune synchronisation globale. Nous proposons une API pour permettre à la couche d'application d'être informée de la phase actuelle du nœud. Dans notre *Deuxième Étude de Cas*, nous présentons une application, un éditeur graphique collaboratif distribué, construit au-dessus du système de stockage clé-valeur, et nous expliquons comment le rendre conscient de sa Phase, à savoir, comment optimiser son comportement en fonction d'une observation en temps réel de la phase à chaque nœud. De plus, nous analysons comment l'application elle-même peut réaliser la réversibilité dans sa sémantique. En utilisant la phase du nœud sous-jacent, l'application fournit une indication à l'utilisateur en ce qui concerne son comportement (actuel et prévu). Ainsi, l'application peut améliorer son comportement par rapport à l'utilisateur, à savoir, l'utilisateur peut comprendre et décider quoi faire dans un environnement de stress élevé.

Nos études de cas montrent l'utilité de la réversibilité pour la construction de systèmes distribués et de la phase pour la construction d'applications au dessus de ces systèmes. La réversibilité donne une fonctionnalité améliorée (interne à l'application) et la phase donne une meilleure facilité d'utilisation (pour l'utilisateur externe). Ainsi, basé sur ces concepts, dans cette thèse nous préparons la voie pour la construction des systèmes informatiques complexes qui peuvent être prouvés corrects, notamment les systèmes distribués et des applications à utilisation améliorée, pour les environnements opérationnels avec un stress arbitrairement élevé.

Sammanfattning

Ett paradigmskifte är på gång inom området distribuerade datorsystem. Från datorer i datacentra är vi på väg mot "edge computing", dvs beräkningen flyttas mot den logiska extremen längst ut i kanten av nätverket. Således, omgivningen för de distribuerade systemen fortsätter att vara mer stressande på grund av inneboende opålitlighet och heterogeniteten i nätets kant. Till exempel, i fallet med mobila och ad hoc-nätverk kan "churn" (nod-omsättningen) vara extremt hög pågrund av nodernas rörlighet, frekventa nerkopplingar / återanslutningar och konfigurationsändringar. Dessutom, i sädana dynamiska miljöer kan systemet möta hög "churn" och nätverkspartitionering på ett återkommande sätt. I denna avhandling föreslår vi en generisk strategi för att bygga distribuerade applikationer som låter dem att överleva, men ändåerbjuda användbar tjänst till användarna i operativa miljöer med hög stressnivå, som dessa.

För att överleva i stressiga miljöer såsom kantorienterade nätverk, behöver systemet ha starka självläkande egenskaper. Vi gör empiriska studier av både beteende och utformning av ett system i miljöer med hög stressnivå. Med andra ord söker vi svar på följande icke-triviala frågor: Kan ett system läka skador på grund av en spänning i omvärlden och återfå sin fulla funktionalitet när stressnivån minskar? Kan vi förutsäga beteendet hos systemet, dvs tillgänglig funktion under stress? Vilka är förutsättningarna; med andra ord, vilka egenskaper bör den helande mekanismen ha för att uppnå dessa mål mot stressiga miljöer?

För att besvara dessa frågor inför och formaliserar vi först begreppen *reversibilitet* och *fas*. Reversibilitet innebär att funktionaliteten hos ett system (uppsättningen av tjänster som det kan erbjuda på ett tillfredsställande sätt) är en funktion (reversibilitetsfunktionen) av sin nuvarande spänningsnivå i dess omvärld och beror inte på stresstillståndets historia. Eftersom stress är ett globalt tillstånd som inte lätt kan mätas med enskilda noder, föreslår vi begreppet *fas* för att approximera reversibilitetsfunktion. Fas är en per-nod egenskap som ger en kvalitativ indikation av systemets funktionalitet som finns i den noden. Fasen kan bestämmas utan ytterligare distribuerad beräkning. I denna avhandling förklarar vi nyttan av dessa begrepp med hjälp av två fallstudier.

Vi bygger på begreppet Strukturerat Overlay-Nätverk (SON), en välkänd metod för att bygga decentraliserade distribuerade system. I vår *Första studie* utvidgar vi ett befintligt SON som härbärgerar ett transaktionsbaserat "key-value-store" så att det blir reversibelt, vilket innebär att systemet kan återfå sin ursprungliga funktion när stressen (t.ex. churn, nätverkspartitionering, det fysiska nätets dynamik etc.) avtar. Vi utvärderar reversibiliteten hos systemet med hjälp av befintliga overlaybaserade underhållsstrategier, nämligen "Correction-on-Change", "Correction-on-Use", "Periodisk Stabilisering", och "Ring Merge". Vi föreslår en ny elastisk underhållsmekanism kallad *Knowledge Base*, för att förbättra villkoren för Reversibiliteten mot ogästvänliga miljöer. Vi identifierar de nödvändiga underhållsstrategierna för att uppnå reversibilitet. Med hjälp av simuleringar visar vi reversibilitet för ett överlappande nät med höga värden för partition, churn och paketförluster. Vi analyserar också interaktionen mellan två stressparametrar, nämligen churn och nätverkspartitionering och identifierar gränserna för samtidig separering och churn, upp till vilket systemet kan uppnå reversibilitet av sig själv. Vi gör generella slutsatser om möjligheten av underhålls strategier för att uppnå reversibilitet.

För att approximera reversibilitetsfunktionen, och därmed beskriva det kvalitativa beteendet hos ett system, introducerar vi begreppet *fas*. Alla noder i en fas uppvisar samma kvalitativa egenskaper, vilka är olika för noderna i olika faser. Eftersom fasen vid varje nod ger en kvalitativ indikation på vilka systemoperationer som finns tillgängliga, tillåter det tillämpningarna, som körs på toppen av systemet att hantera sina beteenden förutsägbart i stressiga miljöer. Vi visar att det finns *reversibla fasövergångar* (dvs en betydande andel av noderna ändrar fas) i systemet som samtidigt upplever stress och visar att vårt koncept fas är analogt med den makroskopiska fasen av fysikaliska system. Vi identifierar empiriskt de kritiska punkterna (dvs när det finns mer än en fas samtidigt i betydande fraktioner av systemet) som observerades i vöra experiment. Vi utökar varje nod i systemet till att vara självmedvetna för att beräkna sin fas utan någon global synkronisering. Vi föreslår ett API för att göra det möjligt för applikationslagret att bli informerat

om den aktuella fasen av en nod. I vår *andra fallstudie* presenterar vi ett användningsfall, en distribuerad samarbetsorienterad grafisk editor, byggd ovanpå vår “key-value-store”, och vi förklarar hur man gör denna *fas-medveten*, det vill säga, optimerar dess beteende i enlighet med en realtids observation av fas vid varje nod. Dessutom analyserar vi hur själva tillämpningen kan uppnå reversibilitet i semantiken på tillämpningsnivån. Genom att använda fasen hos den underliggande noden ger tillämpningen en indikation till användaren om sitt beteende (både det nuvarande och det förväntade). Sålunda har programmet förbättrat sitt beteende i förhållande till användaren, det vill säga användaren kan förstå och besluta vad man ska göra i en miljö med hög stress.

Våra fallstudier visar användbarheten av reversibilitet för att bygga distribuerade system och fasmedvetenhet för att bygga applikationer. *Reversibilitet* ger förbättrad funktionalitet (internt för tillämpningen) och *fasmedvetenhet* ger förbättrad användbarhet (för externa användare). Således, baserat på dessa begrepp, förbereder vi i denna avhandling sättet att bygga bevisbart korrekta komplexa beräkningssystem, särskilt distribuerade system och tillämpningar med förbättrad användbarhet, för godtyckligt stressiga miljöer.

Acknowledgments

This research is partially funded by the Erasmus Mundus Doctorate Programme under Grant Agreement No. $2012 - 0030$ and by the SyncFree project in the European Union Seventh Framework Programme under Grant Agreement No. 609551.

I am deeply thankful to the following people, without the help and support of whom I would not have managed to complete this thesis:

My supervisors Peter Van Roy and Vladimir Vlassov, for their guidance, constant feedback and encouragement throughout this work.

Yves Jaradin and Sébastien Doeraene, for their assistance with the Mozart programming system, in which most experimentation was conducted in this dissertation.

Boris Mejías Candia, for the discussion and assistance with the Beernet system, which is the outcome of his Ph.D. thesis, used for the experiments in this thesis.

Jérémie Melchior, who has collaborated on a paper demonstrating the usefulness of the Phase concept using an application, and who also has used Reversible Beernet for implementing his system, a Distributed Mobile UI (outcome of his Ph.D. thesis), and for his continuous assistance towards making our representative system well defined.

Manuel Bravo and Zhongmiao Li, for collaboration on a paper on the resilience of a Dynamo-style key-value store, Riak-core, in community networks, and for the good discussions.

My former colleague, Nicholas Rutherford for reviewing initial work and valuable discussions.

Vasiliki Kalavri, for sharing her experiences with me, for reviewing and giving useful tips regarding the presentation of papers.

Thomas Sjöland, for his help with my admission to KTH and a translation of a summary of this thesis to Swedish.

The anonymous reviewers of the papers, for providing observant and constructive feedback.

My EMJD-DC colleagues and colleagues at UCL and KTH, especially Navaneeth Rameshan, Amin Khan, Hooman Peiro Sajjad, Paris Carbone, Vamis Xhagjika, Jingna Zeng, Leila Sharifi, Ying Liu, Petar Mrazovic, Christopher Meiklejohn, Seyed Hossein Haeri, and Saranya Natarajan for always being available to discuss ideas, for being supportive and for the good times we spent hanging out together.

The administrative staffs at UCL and KTH, especially Vanessa Maons (UCL), Sophie Renard (UCL) and Sandra Gustavsson Nylén (KTH), for being always helpful and showing way through the complicated administrative path of joint degree.

My family, for their continuous support, their faith in me, their patience and immeasurable understanding.

Louvain-la-Neuve and Stockholm, 11 October 2016

Ruma R. Paul

To My Parents...

Contents

List of Figures

Part I

Introduction

Chapter 1

Introduction

The advent of the Internet has given rise to software systems in which a set of autonomous computers connected through a network, communicate and coordinate their actions as well as share resources. Such *Distributed Systems* are perceived by the users as a single, integrated computing facility with a well-defined goal by the users, i.e., *Distribution Transparency*. There are many dimensions of variability in a distributed system, examples include types of computation units and communication mechanisms among them, timing assumptions on computation and communication speeds, types of failures etc. Among these, timing assumptions and failures (of both network and computing components) are the two dimensions which mostly lead to the breaking of distribution transparency and complicate programming of such systems. Apart from these, the advancement of technology and user demands have triggered an explosion of the population of computing units (often referred as nodes or hosts in a distributed system). As a result, as the number of nodes increases, achieving scalability (i.e., maintaining the same Quality-of-Service (QoS)) is another issue that a distributed system needs to deal with.

It is very difficult to reason about time in a distributed system, especially to pose time bounds on events and communication steps. Distributed systems can be modeled in different ways based on timing assumptions. There are *Asynchronous Systems*, in which there is no time bound of computation and communication, i.e., no timing assumptions can be concluded. Such models are more realistic, as several sources of asynchrony are present in a large-scale network, like the Internet. At the other end of the spectrum, there are *Synchronous Systems*, e.g., LAN/cluster, which have known upper bounds on time to compute and deliver messages. Among these two extremes, there are *Partially Synchronous* distributed systems, about which some timing assumptions can be made, others not, or assumptions can be made statistically, or assumptions hold for arbitrarily long periods of time. In other words, such systems face periods (no known bound to these periods), where timing assumptions do not hold, i.e., the system is asynchronous during those periods; however the system becomes synchronous *eventually* (no knowledge of exactly when). Practical systems are essentially partially synchronous, for example, faulty machines may cause a slow down of processes and network congestion may cause messages to be dropped, thus violating the timing assumptions on computation and communication speed. However, it is relatively easy to define physical time bounds that are respected *most of the time*.

In a distributed system, both node and communication failures may occur, i.e., they deviate from their correct or desirable behavior. Given n nodes connected by communication links, any set of nodes may crash, any communication link or set of links may slow down or fail, and permutations of these failure scenarios can lead to infinite possibilities. One of the reasons that makes distributed systems different (and complex) from non-distributed systems is *partial failures*. Partial failures imply that part of the system (node or network) can fail; however it is desirable that the remaining components of the system continue to operate as expected, i.e., maintains distribution transparency. Thus, handling partial-failure through complete recovery is a key challenge in building reliable distributed systems.

The initial approach for building distributed systems is the simple client-server architecture, in which each node is either a client or a server. Though this is still a popular architecture, the server(s) become(s) a single point of failure and congestion, thus endangers both fault-tolerance and scalability of the system. Due to increasing industrial need of fast Internet connectivity and non-stop operation there was a boom of data-center based systems during the last two decades. However, a paradigm shift is underway in the field of distributed computing systems. We are moving away from data-center-based computing toward *edge computing*, i.e., pushing the computation towards the logical extreme of a network. As a result, such systems face increasing asynchrony, along with failures. The inherent unreliability and heterogeneity of edge networks introduce a crucial challenge for availability, reliability and resiliency for distributed systems as the size of the system grows. The *Peer-to-Peer (P2P)* systems are alternate architecture, and a prime example of edge-computing, which make use of resources available at the edge of a network, and thus has become a popular way of conceiving distributed systems. There are also social factors behind such popularity, as people are willing to connect more to other people and share resources in order to achieve a mutually beneficial exchange. Thus, distributed systems are becoming larger and more complex.

A large-scale distributed system consists of many interacting parts and their overall behavior cannot be predicted in a straightforward way from the behavior of each part. They have many operating modes depending on the environment in which they work and what they are supposed to do. These characteristics of distributed systems inspire approaches to consider them from the point-of-view of *Complex Systems*. In such systems, complex structures or behaviors at macro-level emerge from simple interactions or behaviors of the various subsystems at the micro-level [1, 2, 3]. In a P2P system, it is necessary to identify the macro-level behaviors that emerge due to the interactions and actions of individual peers. This chapter provides a brief discussion on P2P systems and describes the problem addressed in this thesis.

1.1 Peer-to-Peer Systems

The basic idea of P2P systems is the dual client/server role of each node/peer of the system, which allows taking advantage of the resources available at the edge of the network. The first widely known P2P system is *Napster* [4], a file sharing service, which allows users to

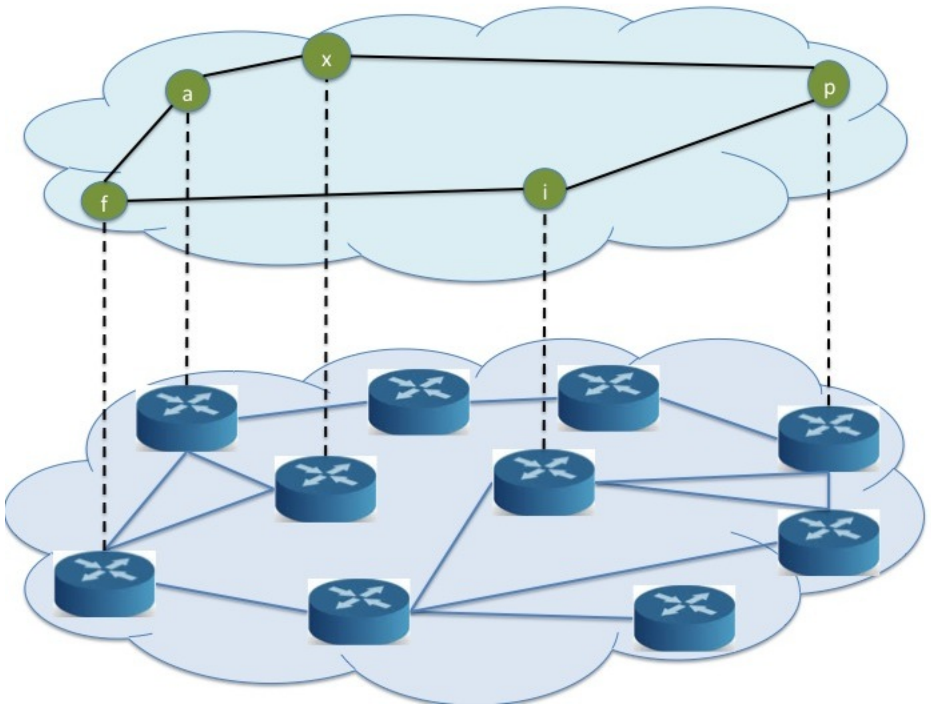


Figure 1.1 – Overlay Network: A P2P System with nodes a, f, i, p, and x forms the overlay network on top of the underlay network

exchange files directly instead of via a server. *AudioGalaxy* [5] and *OpenNap* [6] are two other systems, which belong to the first generation of P2P networks. However, all these systems are not purely peer-to-peer, they are based on mixed architecture and still require a server to provide their services. Peers connect to a server with a query about files. In the reply, the server provides the addresses of the peers storing the requested file. The peer then directly connects to the responsible peers to download the file. In case of server failure, it is not possible to make new queries, but the undergoing exchange of files can continue. Due to the centralization of search operation and administration it was easier to shut-down such systems. Napster ceased its operation in 2001 due to legal issues.

The second generation of P2P systems are the first completely decentralized ones. Gnutella [7] and Freenet [8] are the main two representatives of this generation. These systems do not rely on any server to work. Each peer establishes and maintains some local connections; search queries are forwarded using these connections and replied by the responsible peer(s). Due to such local cooperation of participating nodes an overall network routing view emerges, which is known as an *overlay* network, on top of the underlay network. Usually the underlay network is the Internet, which is used by the overlay network for the routing of messages. Figure 1.1 shows an example overlay network formed by a P2P system with 5 participants.

For the second generation of peer-to-peer networks, the overlay formed is unstructured, i.e. peers connect to each other in a random manner, without any predefined topology. In order to search for a data item, a node sends a query to all of its neighbors. Each node that receives such query forwards it to all of its neighbors. If the query reaches a node that holds the requested data item, it transfers the data item to the querying node. This approach suffers from a high query overhead, as search queries are flooded through the network. This raises concerns about scalability of such systems. Also, if flooding is done with a time-to-live (i.e., the query terminates after a number of steps or hops irrespective of whether the query has reached a node holding the sought data item), there is no guarantee of successful termination of a query. All these problems of unstructured overlay networks have made the way for a structured solution, i.e., *Structured Overlay Networks (SONs)*, the third generation of P2P systems. A structure is induced through the pointers maintained by each peer of the system, i.e. the structure of the emerged overlay is a macro-level property based on the micro-level states of the participating peers. An identifier space is embedded into the graph of a structured overlay, where each node is assigned a unique identifier from this space and also responsible for certain identifiers. Adding structure to Peer-to-Peer networks has provided efficient routing, guaranteed reachability and consistent retrieval of information; however it also introduced the challenge of maintaining the structure. Among all the structures proposed for SONs, the ring topology is the most popular choice. As mentioned in [9], the ring topology is competitive compared to other SONs in terms of reaching any other node in smaller steps and is also the most resilient to failures. Unfortunately, along with these good properties, the temporary inconsistency of the ring structure poses several challenges for correct operations. Many SONs were proposed to gradually improve and circumvent or relax the requirements of a perfect ring for accuracy; *Chord* [10],[11], *DKS* [12], *Beernet* [13], [14] to name a few.

1.2 Self-Management, Fault-Tolerance, Self-Stabilization, and Reversibility

The unreliability and heterogeneity of edge machines, along with different network technologies introduce a crucial challenge for P2P systems: as the size of the system grows, they become more and more difficult to manage. The conventional management becomes very difficult, time-consuming, and error-prone. In order to deal with high-level complexity, enhancing *Self-Management* capacities of the system has no alternative. The term “Self-Management” implies the ability of a system to modify itself, as per high-level management policies, to handle changes in its initial state or environment, without any human intervention. The objectives of self-management are typically classified into four categories: self-configuration, self-healing, self-optimization, and self-protection [15]. Together they are referred in literature as self-* properties.

As already mentioned, partial failures are a reality of distributed systems, which makes *Fault Tolerance* one of the main design objectives of a distributed system. Fault tolerance implies the system behaves in a well-defined manner during failures [16]. In other words, a distributed system is fault-tolerant if it delivers service as per its specification, as long as the faults are from a specified class [17]. Fault tolerance in a system can be masking or non-masking. A masking fault-tolerant system preserves both safety (i.e., something

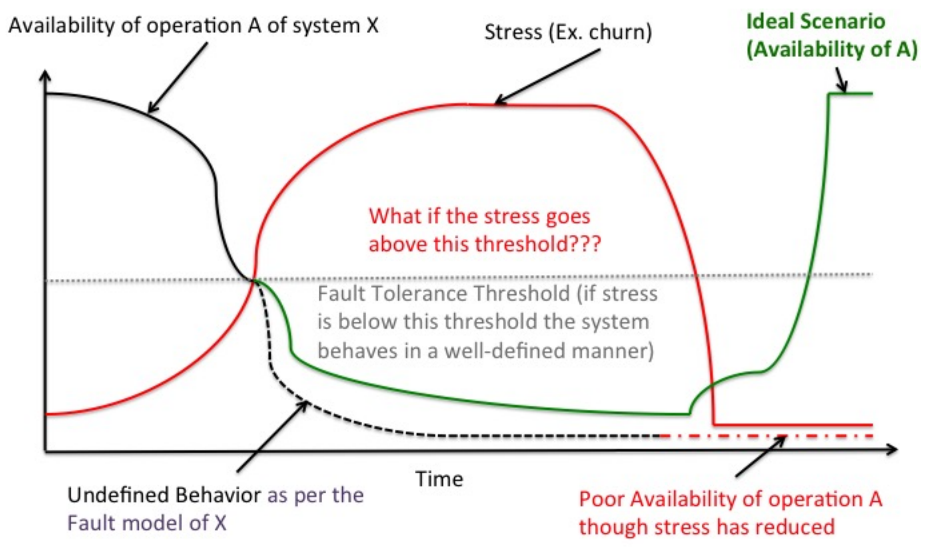


Figure 1.2 – Standard Fault Tolerance and Ideal Scenario

bad never happens) and liveness (i.e., something good eventually happens) properties for a given fault class, whereas a non-masking fault-tolerant system provides guarantees only liveness properties for a given fault class [16, 17].

Recently much work has been conducted on the concept of *Self-Stabilization* [18], [19] as non-masking fault-tolerance for distributed systems. A system is self-stabilizing iff: i) irrespective of the initial state, it will eventually converge to a correct state, and ii) once it reaches a correct state, it is guaranteed to stay in a correct state, provided no fault happens. These two properties enable a self-stabilizing distributed algorithm to recover from transient failures, provided that they do not continue to occur. Here, a transient failure is an event that may change a system's state, but not the behavior of the system [18].

Suppose a non-masking fault-tolerant system X (see Figure 1.2). The system X has invariant P (i.e., P contains the legal states of X) and can tolerate faults from a fault class F with a fault span of T [20]. Here, the fault-span, T characterizes the set of states (called *illegal states*), reachable by faults from F, that X wants to tolerate. Thus, X is intolerant for the states outside of P and T. In Figure 1.2, by "Fault Tolerance Threshold" we mean the intensity of stress, beyond which, faults (from within and/or outside of F) are triggered in the system causing transitions to the states outside of P and T. As long as the intensity of stress is below this threshold, X behaves as per its fault model, i.e., the stress can trigger faults $\in F$, causing transition to states, where safety might be violated (i.e., a certain amount of incorrect behavior is experienced by the user), but liveness is guaranteed, often referred as *graceful degradation*. However, in distributed settings, a system will practically always contain configurations which lie outside of P or T [16]. How to ensure liveness for the system when this happens? Can we ensure well defined behavior (e.g., graceful degradation) of the system, also for the scenarios when the stress intensity goes beyond the threshold, triggering faults form within and/or outside of F and causing transitions to states outside of P and T?

In order to ensure that a system behaves in a well-defined manner while experiencing stress in its operating environment and regains its full functionality as the stress recedes, we introduce the concept of *Reversibility* in this thesis. We define *Stress* as a measure of all the potential perturbing effects of the environment on the system, which includes both faults and other nonfunctional properties such as communication delay and bandwidth. Reversibility of a distributed system means that the system's functionality depends only on the current stress of the operating environment and not on the history of stress the system has experienced in the past. Reversibility generalizes standard fault tolerance with nested fault models. The nesting corresponds to a directed acyclic graph, where each node is a fault model and each edge is a transition from one fault model to the next. When the fault rate goes outside the scope of one model then it is still inside the scope of the next model. Also, reversibility is only concerned about stress imposed on the system, where the stress might or might not cause fault in the system. For example, consider churn (i.e., nodes failing and being replaced by new correct nodes) as the stress. A low churn might not trigger any fault in the system, however churn causes stress for the system to work less efficiently. Also, reversibility does not assume anything about system state. It only guarantees that the system can repair itself to provide full (and correct) functionality after being subject to arbitrary stress. Reversibility is more useful in practice comparing to self-stabilization. A self-stabilizing system survives any temporary perturbation of its internal state and returns to a valid state when there are no perturbations; whereas reversibility gives information about functionality even during nonzero stress.

1.3 Problem Definition

Distributed applications break down when there are too many node or communication failures. Typically, such applications revert to an "offline mode" with reduced functionality in this case. This is sometimes acceptable for applications that have a client/server architecture, such as mobile applications that depend on a data center. The data center remains a single point of failure. However, this is now changing as the Internet is becoming more and more decentralized: data centers are increasing in number and come in various sizes. Also, the technological advances and always-increasing user demands of better QoS and scalability of distributed applications are preparing the way for a paradigm shift in the field. We are moving away from data-center-based computing toward edge computing, i.e., pushing the computation towards the logical extreme of a network: Mobile edge computing, cooperative distributed peer-to-peer ad-hoc networking, Internet of Things (IoT), Wireless Sensor Networks — to name a few of the wide range of technologies covered by edge computing. The inherent unreliability and heterogeneity of edge networks introduce a crucial challenge for distributed systems as the size of the system grows. Applications running on such infrastructures need to have a decentralized architecture that is resilient to failures (of both nodes and communication). Ideally, the application should survive with partial functionality during arbitrary system failures and recover its full functionality when the underlying system is restored. This is not just a fringe case: mobile and ad hoc networks, for example, have this kind of failures. Even supposedly stable parts of the Internet have peaks of unstable behavior.

Due to the inherent complexity introduced as a result of decentralization, large-scale distributed systems, in particular P2P systems have brought forward many non-trivial challenges. These systems face many problems, especially when they are stressed beyond where their behavior is a straightforward extrapolation of the behavior of their parts, i.e. the emergence of macro-level behavior. The experience of all who have attempted to build such large-scale distributed applications is that they are very difficult to get right: they require continual babysitting by teams of specialists to keep them running. As Kephart et al. mentioned in [15], current complex computing systems require skilled IT professionals to install, configure, tune, and maintain; and as they pointed out the complexity of heterogeneous Internet-wide computing systems appears to be approaching the limits of human capability. Also, as pointed out in [21], reliability issues are common in modern distributed systems. Even within a data center failure is frequent [22], and while supporting Internet scale services; growing frequency of faults is an alarming concern [23]. Ian Wilkes shared his experience of working on Second Life (a highly complex virtual world, incorporating the features of Web services, online games, 3D modeling and programming tools, IM and VOIP etc.) [24]: he shared the challenges those were brought forward while scaling up. As he pointed out it is a major challenge in the real world to discover the complete set of errors that are possible, or likely, or common and very easy for developers to lose track of errors. He has also shared the over-paging problem and challenges faced by system operations people as the system scales up.

The reason behind such experiences of building and maintaining large-scale distributed systems is that, like all highly available systems, these systems are designed to operate within a specific failure model and threat model and their behavior is typically undefined outside these models. The situation becomes worse for a distributed application running on top of a system, whose behavior is non-deterministic or even unknown for a given scenario. This implies non-trivial questions: what behavior should the application layer expect from the underlying distributed infrastructure in a given scenario? Under what conditions should a system exhibit resiliency against arbitrary stressful environments? Is it possible for the system to give useful information, regarding its behavior, to the application layer so that the application can also manage its behavior while experiencing stress in its environment? Existing literature lacks systematic and in-depth studies and analysis to reach conclusive answers to these questions. In this thesis, we explore and investigate about the answers to these questions.

1.4 Hypothesis

The problem we investigate in this thesis is to build distributed systems, that: i) are able to survive with partial functionality during the stress and recover their full functionality when the stress recedes; ii) have well-defined behavior, i.e., predictable in their complete operating space; iii) can provide useful information, in an efficient manner, to the application layer such that, using that information the applications running on top can manage their behavior as per application-level semantics while experiencing stress.

The standard approach, based on redundant fault tolerant algorithms, has reached its limit to provide ultimate solution to this problem. The problem can only be solved by a new approach, which generalizes standard fault tolerance with multiple nested fault models. We propose an approach to build applications that are able to survive arbitrary failures, providing reduced but predictable functionality in that case, and when the failures go away the application fully recovers its functionality. In other terms, to design systems such that a change in operating conditions can cause a qualitative change in system behavior according to a well-defined transition instead of operating unpredictably. Our goal is to design the system to work in a well-defined manner for the complete operating space, i.e., for all possible operating conditions including extremely inhospitable ones. We define an "inhospitable environment" as one in which certain stress parameters (such as churn, node failure rate, communication delay) can potentially reach high values and temporarily increase without bound. The goal is to build systems that are both predictable (hence, useful in practice) and reversible (hence, capable to survive) in these environments. This is important for three essential reasons. First, for practical system design it is important to explore highly stressful environments, since even systems running in so-called "stable" environments will have peaks of high stress. Second, it can open new venues for application design, such as mobile and ad hoc networks, for which current fault-tolerance techniques are insufficient. Third, it is important for scientific reasons, to understand what happens in highly inhospitable regimes.

1.5 Research Methodology

We have conducted empirical research in this thesis in order to investigate about the answers to the questions presented in Section 1.3. We have done three case studies: (1) in our first case study we investigate the design and behavior of a ring-based structured overlay network, Beernet [13], which hosts a transactional key-value store, such that the system is able to survive against arbitrary stressful environments and provide qualitative indication of available functionality at any instant; (2) our second case-study, using a distributed collaborative graphic editor, DeTransDraw [13], shows how the applications can be designed such that using the information provided by the underlying system they can improve their behavior with respect to stress; (3) in our third case study, we apply our approach to a second system called Antidote [25], which is a causal+ consistent transactional key-value store implemented on a consistent hash-based ring overlay. The system used in our first case study, Beernet, is a representative member of a wide class of systems, namely a Chord-like [11] SON extended with replication and transactions. This class of systems constructs and maintains a ring topology and can be described following the reference of [26]. We have chosen ring-based overlays, because the ring is competitive with other SON structures in terms of routing efficiency and failure resiliency[9]. As per data consistency guarantee, Beernet supports strong consistency model. *Strong consistency* ensures that only one consistent state can be observed: when queried for a data item, all replicas return the same value [27], which is not always the fact with *eventual consistency*, which guarantees that if there is no new update to a data item, eventually all accesses to that item will return the last updated value [28]. From correctness perspective strong consistency is a stronger guarantee than eventual consistency. However, strong consistency can cause high-latency, whereas eventual consistency is used to achieve high availability. We have used a system supporting strong consistency for most of the work conducted in this thesis due to simplicity: this thesis investigates about survival of a system in high-stress environment and consistency is orthogonal to our research objectives. Eventual consistency introduces a new level of complexity. To get the crispest results, using strong consistency is the right choice since it is easier to program. However, we have conducted some preliminary experiments using a system (Antidote) which supports eventual consistency in our third case study. This can be extended in our future work, and also investigation of whether (and how) a particular consistency model has any impact on the reversibility of the system can be an interesting future research direction.

Our goal is to build distributed systems, which can survive arbitrary stress. So, as part of our case-studies, we have investigated different stress elements. We have followed incremental approach during our investigations: i) we explore one stress parameter (with zero/constant value of other stress parameters); ii) we investigate interaction between stress parameters; iii) we analyze how the system can direct the application to manage their behavior in stressful environments. In our experiments we have mostly used the simulated environments (except for our third case study) in order to have an in-depth understanding of the behavior and resiliency of a system against a particular stress. For our third case-study the experiments are conducted in a realistic commercial infrastructure.

1.6 Thesis Contributions

There are two main objectives of this thesis: to investigate designs and to study the behavior of complex computing systems, in particular a class of Structured Overlay Networks (SONs) defined by the reference architecture of [26], in inhospitable environments. Also, this thesis explores the design of applications, running on top of such systems, so that they can survive, yet able to provide useful service in stressful operating environments. In this section, we briefly summarize the main contributions of this thesis.

1.6.1 The Concepts of Reversibility and Phase

We introduce the concept of *Reversibility* to ensure the complete recovery of the system, i.e., to regain full functionality after being subject to arbitrary stress. Reversibility also ensures that the system behaves in a well-defined manner during the stress. We define a system to be *Reversible* if the set of operations it provides is a function (will be referred to as the *reversibility function*) of its current stress and does not depend on the history of the stress. Reversibility generalizes standard fault tolerance with a spectrum of fault models. When the fault rate goes outside the scope of one model then it is still inside the next model. However, stress is a global condition imposed on the system that cannot easily be measured by individual nodes. So, in order to approximate the reversibility function, we introduce the concept of *Phase*. The second usefulness of the phase-concept is that it helps the applications to manage their behavior in stressful environments. In contrast to stress, *Phase* is a per-node property that gives a qualitative indication of what system operations are available at each node, given the current stress. Thus, making the phase available to the application layer allows applications to improve their behavior with respect to stress. Each node determines its phase independently and no extra distributed operations are necessary. We define the *Phase Configuration* of a system as the vector of phases of all nodes in the system. Both phase and phase configuration are a function of time. We approximate the reversibility function using the *phase function*, i.e., we say that the set of available operations of the system is a function (aka, the phase function) of its current phase configuration.

1.6.2 Investigation of Stress

We extend an existing Structured Overlay Network (SON) to be reversible against various stress elements. For this purpose, we first identify the stress elements that causes the operating environment of an overlay network to be inhospitable. In other words, we organize the operating space of a SON. In order to achieve reversibility of a system, complete self-healing is crucial. This can be achieved by the maintenance strategy of an overlay. A *Maintenance Strategy* maintains the structural integrity of a SON while peers go offline or network connections fail. We organize the overlay maintenance strategies using a *Efficiency* $\leftrightarrow$ *Resiliency* spectrum. We demonstrate a reversible SON and conditions to achieve the reversibility against churn (i.e., nodes failing and being replaced by new correct nodes), network partitioning (i.e., communication failure in the physical network causes logical partitioning of the overlay), network dynamicity due to packet-loss (i.e., packets are dropped in the physical network). We show that both efficient and resilient maintenance are required to make the system reversible. Further we investigate the interaction between two stress elements: churn and network partitioning. We identify the limits of the combination of partitioning and churn, up to which the system is able to achieve reversibility by itself, using a model of how the partitions diverge with time (the Stranger Model). Also, as part of analyzing reversibility of the system against various stress sources, we perform a comparative analysis of self-healing of different maintenance strategies in inhospitable environments.

1.6.3 Building Applications

We demonstrate an approach to compute the phase function efficiently using our representative system. Each node computes the phase function assuming that the rest of the system is in the same phase as itself, thus avoiding any distributed operation or resource consumption. We identify different phases and sub-phases in our representative system and explain the semantics of those in terms of available functionality. Thus, we demonstrate how we approximate available functionalities of our system at each node based on its current phase. We experimentally demonstrate *Reversible Phase Transitions* in a SON while facing churn and packet-loss. We illustrate self-awareness at each node and how a self-aware node determines its current phase in our representative system. We propose an API to expose the current phase of a node to the application running on top. As the phase of the underlying node gives indication of available functionality, this allows the applications to manage their behaviors predictably in stressful environments. Finally, we demonstrate *Phase-Aware* application design using a use-case application, DeTransDraw (a real-time collaborative graphic editor), and its evaluation. We evaluate self-adaptive and self-optimization behaviors of the application against simulated churn, delay, and partitioning at the physical network level. We show how the application can be predictable for the user regarding current and expected functionality changes due to stress in its operating environment. We establish reversibility at the application-level as well as the SON-level.

1.7 Related Work

In this section we briefly outline the state of the art relevant to the research objectives of this thesis, in order to place our contributions in the context of the existing works. The detailed discussion about the existing literature is presented in a distributed fashion in this thesis, i.e., each chapter presents a discussion of related works relevant to the topics covered in that particular chapter.

The existing works in the literature mostly validates the operations of distributed systems, in particular structured overlay networks (SONs) in usual or some specific (e.g., simultaneous node failures, interleaved join and failures) scenarios, showing the behavior of the system for a limited fault model. The SONs exhibit certain fault-tolerance, self-organizing and self-healing properties. The state of the art mostly validates these inherent properties of a representative SON, where a particular stress remains under a certain limit. As a result, in all these works, the stress causes transitions to the states within the fault span of the particular fault models used. Churn is the most studied stress in the literature, as for a peer-to-peer system joining and leaving or failure of nodes are usual events. Several experimental and analytical works are found in the literature that assess performance or dependability of SONs under churn using various metrics [29, 30, 31, 32, 33, 34, 35, 36]. Recently several works have addressed physical network partition as the stress, and proposed merge algorithms once the partition ceases [37, 38, 39, 40, 41, 42]. There are also works which evaluate SONs by varying environmental parameters like network topology, large link delays, link loss rates, amount of application traffic [29, 43, 44, 36]. However, current literature lacks an effort that explores the entire operating space of a SON, identifies various sources that can cause the operating environment of a SON stressful, and does an in-depth study of the behavior of the system throughout that space. To our knowledge, no previous work exhibits reversibility for a system at any level (e.g., connectivity-level, data-level), by complete self-healing, for bursts of high stress (as investigated in this thesis), and identify the implications when multiple stress interacts. With increasing decentralization, and on the verge of a topological migration towards edge computing in the field of distributed systems, it has become necessary to ensure survival of complex computing systems, such as SONs, against arbitrarily stressful environments, define what survival means in terms of available functionality of the system while experiencing stress and regaining full functionality as the stress diminishes. The research conducted in this thesis is a step towards this direction.

The other research objective investigated in this thesis is to explore how a reversible system can give useful information to the application running on top, so that using that information the application can improve its behavior in stressful environments, achieve reversibility, and be predictable to the user. For this purpose, we introduce the concept of phase to approximate the overall functionality of the system at each node. We define phase by drawing an analogy with the thermodynamic systems. In order to further strengthen the analogy we conduct experimental studies, and describe our representative system from thermodynamics perspective. The vision of *Organic Computing*, i.e., analyzing, describing and designing large-scale distributed systems by drawing analogy with physical or biological systems is comparatively a new research direction and has been gaining momentum in recent years due to increasing complexity of distributed systems. There are several analytical works in the literature, e.g., [31, 33, 45, 46], that apply the approaches and concepts from statistical mechanics and physics to analyze and describe the behavior of SONs. Apart from these, there are visionary articles and papers that present ideas, insights and characterization of analogy of large-scale distributed systems with physical systems [47, 48, 49]. Our approach and empirical studies extend this analogy further, and open new venues of research towards this direction.

1.8 Publications

The work presented in this dissertation is the result of incremental progress made through discussions and presentations in several conferences and workshops.Several of the results presented here have also been published at or submitted to conferences. In this section we present the publications relevant to this thesis.

- **Conference** *"An Empirical Study of the Global Behavior of A Structured Overlay Network"*. Ruma R. Paul, Peter Van Roy and Vladimir Vlassov. IEEE International Conference on Peer-to-Peer Computing, 8-12 September 2014, London, England. This publication has laid out the background of the research conducted in this thesis. The contributions of this paper are: organization of the operating space of a structured overlay network (SON) using stress elements, presentation of a self-adaptable eventually perfect failure detector and evaluation of its Quality-of-Service (QoS), augmentation of Beernet's replica management with a protocol for better availability and consistency of replica sets, definition of the behavior of a SON in terms of a set of parameters and presentation of the behavior of Beernet along one stress, namely Churn. In this thesis, Chapter 4 describes the organization of the operating space of SONs using stress elements and behavioral metrics of a SON. The algorithms for failure detection and replica management are presented in Chapter 5.

- **Conference** *"Interaction Between Network Partitioning and Churn in a Self-Healing Structured Overlay Network"*. Ruma R. Paul, Peter Van Roy and Vladimir Vlassov. IEEE International Conference on Parallel and Distributed Systems, 14-17 December 2015, Melbourne, Australia. This paper is focused on our investigation of network partitioning and interaction between network partitioning and churn. As part of this we have presented a new resilient maintenance, Knowledge Base, in order to improve conditions for reversibility in the case of combined network partitioning and churn. The maintenance using knowledge base in discussed in detail in Chapter 3, whereas Chapter 7 and Chapter 8 present the results of our investigation of network partitioning and interaction between network partitioning and churn.

- **Conference** *"Reversible Phase Transitions in a Structured Overlay Network with Churn"*. Ruma R. Paul, Peter Van Roy and Vladimir Vlassov. International Conference on Networked Systems (NETYS), 18-20 May 2016, Marrakech, Morocco. This paper formally introduces the concepts: Reversibility and Phase, presents the results of our investigation of churn, identifies different phases in our representative system, empirically demonstrates reversible phase transitions in the system as the stress varies and proposes the phase API. In this dissertation, the concepts are discussed in detail in Chapter 2, Chapter 6 presents the results of our investigation of churn, and Chapter 10 is focused on the phases and phase transitions in our representative system.

- **Conference (Submission)** *"Designing Distributed Applications using A Phase-Aware, Reversible System"*. Ruma R. Paul, Jérémie Melchior, Peter Van Roy and Vladimir Vlassov. 18th International Conference on Distributed Computing and Networking (ICDCN 2017), 4-7 January 2017, Hyderabad, India. In this paper we have refined the concept of Reversibility and introduced two different forms of reversibility. Also, we have defined how a node efficiently determines its current phase in our representative system without any global synchronization. We have empirically demonstrated phase-aware application design using a use-case and evaluated the enhancements due to such design aspects. Chapter 11 in this thesis is focused on computation of phase at each node and phase-aware application design.

- **Licentiate Thesis** *"An Empirical Study of the Global Behavior of Structured Overlay Networks as Complex Systems"*. Ruma R. Paul. KTH Royal Institute of Technology, 20 October 2015, Stockholm, Sweden. The licentiate thesis includes parts of the results presented in this thesis, especially the contributions discussed in Chapter 5, Chapter 6, Chapter 7 and Chapter 8 are also presented in the licentiate thesis.

- **ICTEAM Scientific Highlights 2016 (Research Report)** *"An empirical study of the global behavior of structured overlay networks as complex systems"*. Ruma R. Paul and Peter Van Roy. Université catholique de Louvain, Louvain-la-Neuve, Belgium. A one-page summary and vision of the research work conducted in this thesis.

- **Conference (Abstract)** *"Building Distributed Applications for Stressful Environments using Reversibility and Phase-Awareness"*. Ruma R. Paul and Peter Van Roy. The Conference on Complex Systems, 19-22 September 2016, Amsterdam, The Netherlands. A 300-words abstract (peer-reviewed) describing the vision and the research work conducted in this thesis.

1.9 Thesis Organization

The rest of this thesis is organized in parts and each part has several chapters. As mentioned in Section 1.7, we present the existing works in the literature, relevant to this thesis, in a distributed fashion, i.e., at the end of each chapter we discuss about the corresponding related works. In this section, we provide the brief outline of this thesis.

- Part II — Foundations: In this part, we lay out the foundation of this thesis: the concepts (reversibility and phase), representative systems used for case studies, maintenance in these systems to achieve reversibility, the stress sources. Also, our contributions to enhance our representative system: QoS-aware eventually perfect failure detection algorithm and data migration protocol for replica management.
  - Chapter 2 presents the concepts of *Reversibility* and *Phase*, and their relation;
  - Chapter 3 describes our representative class of complex systems, namely a class of Structured Overlay Networks (SONs), our representative system used for experimentation, organizes the maintenance strategies of SONs, and discusses existing and our contribution in maintenance strategies of SON;
  - Chapter 4 categorizes the operating space of overlay networks considering five stress sources. It presents a set of metrics to define the behavior of a SON;
  - Chapter 5 describes an eventually perfect QoS-aware failure detection algorithm and an optimistic, low cost data migration protocol for replica management.
- Part III — Stress Investigations: In this part, we investigate about several stress elements individually. We assess reversibility of our representative system against churn, network partitioning and network dynamicity due to packet loss and identify the preconditions, i.e., properties of maintenance required to achieve reversibility when a system experiences such stress in its operating environment. Also, we explore interaction between two stress elements: churn and network partitioning and identify the limits up to which the system can achieve reversibility by itself.
  - In Chapter 6 we investigate churn and the design of reversible systems that can sustain extremely high churn rates;
  - Chapter 7 presents our investigation on network partitioning;
  - In Chapter 8, we explore the interaction between the two stress sources, namely network partitioning and churn;
  - Chapter 9 investigates stressful operating environments for an overlay due to packet-loss in physical network.
- Part IV — Practical Applications : In this part, we investigate about the implications of our concepts towards designing practical systems. We propose an approach to approximate the available functionality of a system given the current stress and apply this approach to our representative system. We demonstrate phase-aware application

design using a use-case. We also apply our concept of reversibility to another system and conduct experimentation in real-world scenario.

  - In Chapter 10 we describe our approach of approximating the set of available operations of our representative system at any instant, we describe all phases and sub-phases we have identified in our representative system, and we study phase transitions in a SON due to stress;
  - Chapter 11 describes how we have extended a peer to become self-aware so that it can determine its current phase, we present the API to expose the current phase of a node to the application, and we demonstrate a phase-aware application design using a use-case application, DeTransDraw (a real-time collaborative graphic editor);
  - In Chapter 12, we apply our concept of Reversibility to a second use-case, a causal+ consistent transactional key-value store implemented on a consistent hash-based ring overlay.

- Part V — Conclusions: In this part, we summarize this thesis and explore some future directions of research as outcomes of this thesis.
  - Chapter 13 summarizes the contributions of this work;
  - In Chapter 14, we discuss future directions of research relevant to the work presented in this thesis.

Part II

Foundations

Chapter 2

Concepts

The goal of this thesis is to explore how to build distributed systems that are able to survive, and yet able to provide predictable functionality in inhospitable environments. An inhospitable operating environment with respect to a system is the one in which certain sources of stress (such as churn, node failure rate, communication delay) can potentially disrupt the functionality of the system and the intensity of stress can temporarily increase without bound. We define stress as a measure of all the potential perturbing effects of the environment on the system, which includes both faults and other nonfunctional properties such as communication delay and bandwidth. In order to incorporate the above mentioned objective and relate precisely the current stress with the current available functionality provided by the system, we introduce the concept of *Reversibility* [50]. The term "Reversible" implies that the system is able to regain its original functionality as the stress recedes. In other words, the set of operations provided by a reversible system is a function (we call this the *reversibility function*) of current stress in the operating environment and does not depend on the history of stress the system has experienced in the past.

With standard fault tolerance, we can specify the behavior of a system for a given fault model, i.e., when the stress causes failures that stay below a threshold. However, what is the expected behavior of the system if the stress goes beyond that threshold? How to specify a system that survives arbitrary stress with limited functionalities? To answer these questions, we need a concept that generalizes standard fault tolerance, we propose such a concept, called Reversibility.

In order to compute the reversibility function we need a measurement of stress as a function of time. But, stress is a global condition imposed on the system that cannot be easily measured by individual node. In order to approximate the reversibility function at each node we propose the concept of *Phase* [50]. In contrast to stress, phase is a per-node property that gives a qualitative indication of what system operations are available at each node, given the current stress. In this chapter, we present a *Formal Definition of Reversibility* and discuss how reversibility is related to Fault Tolerance and Self Stabilization. This chapter also describes the *Phase* concept and its relation with reversibility.

2.1 Reversibility

Suppose a distributed system running on n nodes providing a specific set of services. From time t to $t+T$, the system experiences external stress, e.g., k nodes crash and j nodes join the system or a system partition due to a connectivity problem of the underlying physical network. Can we ensure that the system will eventually regain its full functionality after time $t+T$? Does the system have a well-defined behavior during the interval $[t, t+T]$? We introduce the concept of *Reversibility* to answer both of these questions. Reversibility implies the system's functionality depends only on the current stress in its operating environment and not on the history of stress.

Definition Suppose a function $S(t)$, which returns the system stress as a function of time, in some arbitrary but well-defined units. The stress is a global condition imposed on the system. Individual nodes will normally not know the stress, i.e., it is not usually known as one-node property. For example, $S(t)$ can explain how the system is partitioned as a function of time. Also, $S(t)$ can be any complex data-structure (e.g., a vector or dictionary, one entry per node) without losing generality. In Chapter 4, we have organized the operating space of a system using five dimensions, which are basically five sources of stress, that can cause inhospitable operating environment for a system.

A system is *Reversible* if there exists a *total* function $F_{rev}(S(t))$ (we call this as the *reversibility function*) such that the current set of available operations of the system (i.e., the functionality of the system), $Op_{avail}(t) = F_{rev}(S(t))$, and when $S(t) = 0$ (Here, 0 is used as a symbol to indicate "no stress"), the system provides full functionality at all nodes. Here, Op_{avail} can be a vector, one entry per node and each entry can be a set of available operations at that particular node.

We define Op_{total} as the set of overall functionality of a system. If the system is reversible, when $S(t) = 0, \forall p, Op_{avail}[p](t) = Op_{total}$, where p is the conceptual component (often referred as node) of a distributed system and $Op_{avail}[p]$ denotes the set of available operations at p. For nonzero values of stress, $\forall p, Op_{avail}[p](t) \subseteq Op_{total}$.

Two forms of Reversibility From the above definition, we can identify *two* different forms of reversibility.

- Weak Reversibility: A system is *Weakly Reversible* if for all t such that $S(t) = 0$, the system provides full functionality at all nodes. All operations are guaranteed to complete if $S(t) = 0$ for a sufficient time interval to cover the operation.
- Strong Reversibility: A system is *Strongly Reversible* if there exists a function $F_{rev}(S(t))$ such that the set of available operations of the system is equal to $F_{rev}(S(t))$. An operation is *available* for a given stress if the operation will eventually succeed (i.e., it will fail only a finite number of times if tried repeatedly, and then succeed).

For *Strong Reversibility*, if stress starts at any value s_1 and changes to s_2, and then goes back to s_1, then the original functionality that was available at s_1 is regained. For *Weak*

Reversibility, s_1 is always 0, and nothing is said for nonzero values of stress. In Section 2.2, we propose an approach to approximate the reversibility function.

Reversibility and Fault Tolerance *Reversibility* is a related but different property than *Fault Tolerance*. Fault tolerance implies the system behaves in a well-defined manner during failures [16], i.e., a fault-tolerant system is resilient for a given fault model, but its behavior outside that model is undefined. In order to specify a system that survives for arbitrary stresses with limited functionalities, we would have to define a separate fault model for each of the limited functionalities provided. By introducing reversibility we generalize standard fault tolerance with nested fault models. The nesting is more complicated than placing each fault model inside exactly one fault model (for example, like Russian dolls [51]). The complication arise due to the fact that, in a system with several stress parameters, there will often be several ways to go out of a fault model and into another one. So, we conjecture the nesting as a directed acyclic graph, where each node is a fault model and each edge is a transition from one fault model to the next. As a result, when the current stress causes fault rate to go outside the scope of one model (i.e., outside its fault span) then it is still inside the scope (i.e., within the fault span) of the next model.

Reversibility and Self-Stabilization Recently much work has been conducted on *Self Stabilization* [18], [19], as a non-masking fault-tolerance for distributed systems. Unlike self stabilization, reversibility does not assume anything about system state. A self-stabilizing system survives any temporary perturbation of its internal state; it returns to a valid state when there are no perturbations, whereas reversibility is more useful in practice as it gives information about functionality even during nonzero stress. A self-stabilizing system is reversible, but a reversible system is not necessarily self-stabilizing. However, self stabilization implies weak reversibility. Weak reversibility may imply self Stabilization in some cases, if the values of $S(t)$ can cause the system's state to be any arbitrary value.

2.2 Phase

In order to approximate the reversibility function we introduce the concept of *Phase*. In contrast to stress, which is a global condition that cannot easily be measured by individual nodes, the phase is a local property of a node. A *Phase* is a subset of a system for which the qualitative properties are essentially the same. Each node determines its phase independently and no extra distributed operations are necessary. Also, phase is a useful concept for designing stress-aware distributed application services. A system that provides significant functionality at zero or low stress to the application running on top, will no longer be able to do so as the stress gets intensified. Applications that rely on those functionalities, that get disrupted due to stress, will no longer be able to use them. We want these applications to continue running nevertheless, with predictable behavior even with reduced functionality. For this purpose, the application needs to be informed about the qualitative changes of the provided functionalities by the underlying system. This can be achieved by exposing the phase of each node: through phase the system can provide qualitative indication to the application regarding what system operations are available.

Definition A *Phase* is a subset of a system for which the qualitative properties are essentially the same. For this definition we consider a system as an aggregate entity composed of a large number of interacting parts. Different parts can be in different phases, depending on the local environment observed by the part. Boundaries between phases in a system can be either sharp or diffuse. As a node experiences changes in its local environment, it changes its phase independently of the other nodes. A *Phase Transition* occurs when a significant fraction of a system's parts changes phase. This can happen if the local environment changes at many parts. A *Critical Point* occurs when more than one phase exists simultaneously in significant fractions of a system. Phases are observed in many large systems. They are well-understood in physical systems consisting of large numbers of atoms or molecules (e.g., macroscopic amounts of water, in solid, liquid or gaseous phases) [52], but they can also be observed in computing systems.

We consider a system, $S = \{S_1, \ldots, S_n\}$, where each $S_i : 1 \leq i \leq n$, is an interacting part of S. The system is partitioned into k subsets, such that: (1) $P_1 \uplus P_2 \uplus \ldots \uplus P_k = S$, (2) For any $P_i : 1 \leq i \leq k$, qualitative properties are the same $\forall S_x \in P_i$, (3) For $P_i, P_j : i \neq j$, qualitative properties are different, i.e., if F is a function of qualitative property, then $\forall_{i,j:i \neq j} \implies \forall S_x \in P_i, \forall S_y \in P_j : F(S_x) \neq F(S_y)$. We can say each $P_i : 1 \leq i \leq k$ is an observed phase in S. A *Phase Transition* at system level occurs when a significant fraction of a system's parts changes phase. This can happen if the local environment changes at many parts. A *Critical Point* occurs when: (1) $k > 1$, (2) $\forall i, 1 \leq i \leq k, \mid P_i \mid \gg 1$.

Approximating the Reversibility Function We have defined the *Phase* P_i at each node i to be a well-defined local property of the node. We define the *Phase Configuration* of the system to be the vector $P_c = (P_1, P_2, P_3, \ldots, P_n)$. Both phase and phase configuration are functions of time. If the phase configuration contains enough information, then we can define Op_{avail} (see Section 2.1) in terms of it: $Op_{avail}(t) = F_{rev}(S(t))$ $\approx F_{phase}(P_c(t))$. $F_{phase}(P_c(t))$ (we call this as the *phase function*) is a total function: $\forall P_c(t), F_{phase}(P_c(t)) \mapsto Op_{set}$, here, Op_{set} is an approximation of Op_{avail}, thus similar to Op_{avail}, Op_{set} is also a vector, one entry per node and each entry is a set of available operations at that particular node. In Chapter 10 we show how the phase function can be computed efficiently using a representative system.

Chapter 3

Representative Systems

For practical reasons, we have chosen a particular class of complex systems to conduct our study in this thesis. We build on the concept of *Structured Overlay Network (SON)*, a well known approach for building decentralized distributed systems. As per the reference architecture, proposed in [26], a class of SONs, namely ring-based SONs are our representative complex systems. For the practical experiments, we have chosen Beernet [13], a ring overlay with properties typical of this space. Beernet hosts a transactional key-value store. Beernet is a version of Chord [11], unlike Chord, Beernet has a correct lock-free join/leave operation and in this thesis we have extended Beernet to become reversible. In this chapter, we describe key aspects of our chosen class of SONs and Beernet.

3.1 Ring Structured Overlay Networks

We have chosen ring-based overlays as our representative systems, because the ring is competitive with other SON structures in terms of routing efficiency and resiliency to failures [9]. Examples of ring-based SON include Chord [11], Chord# [53], SkipNet [54], DKS [12], Koorde [55], P2PS [56], Beernet [13], Mercury [57], [58], EpiChord [59], Accordion [60], Symphony [61].

According to the reference model proposed in [26], there are six key design aspects, that can characterize any overlay network. Here, we provide a brief description of this design space, along with the specification of ring overlays. Our contributions can be generalized for other overlays having similar key aspects in the design space.

- Choice of Identifier Space: virtual identifier space I, having some closeness metric $d : I \times I \to \mathbb{R}$. For ring overlays identifier space is a subset of $\mathbb{N}$, of size N, with $d(x, y) = (y - x) \, mod \, N$.

- Mapping to the Identifier Space: $F_P : P \to I$ associates peers with a unique virtual identifier from I and $F_R : R \to I$ associates resources with identifiers from I. For ring overlays, F_P can be a uniform hash function or some random function, it also can be order preserving, as in DKS. A virtual identifier is assigned to a peer when it joins the overlay and this mapping remains static. F_R is similar to F_P for ring overlays, usually it is a uniform hash function, which distributes resources uniformly in the identifier space, thus providing implicit load balancing.

- Management of the Identifier Space: $M : I \rightarrow 2^P$ associates with identifier of a resource r, $i = F_R(r) \in I$, the set of peers managing r. Each peer p is responsible for the set $M^{-1}(p)$ of identifiers. For ring overlays, a peer with virtual identifier p is responsible for the interval $(predecessor(p), p]$. In these approaches, the responsibility of a peer may dynamically change due to churn in the overlay.

- Graph Embedding: a directed graph, $G = (P, \varepsilon)$, where P is the set of peers and ε denotes the set of edges. A neighborhood relationship $N : P \rightarrow 2^P$, for a peer p, $N(p)$ is the set of peers with which p maintains a connection. The graph formed by ring overlays complies with Kleinberg's small-world principles [62], thus belongs to the special class of routing-efficient, small-world networks. In these overlays, each peer p perceives the identifier space to be partitioned into $\log(N)$ partitions, where each partition is k ($k = 2$ for Chord) times bigger than the previous one. The routing table of p contains $\log_k(N)$ connections to some nodes from each partition. Also, for resiliency purpose, each node of ring overlay maintains a *successor list*, which looks ahead on the ring. The successor list of p contains connections to $\log_k(N)$ consecutive nodes in clockwise direction.

- Routing Strategy: a non-deterministic function $R : P \times I \rightarrow 2^P$, which at peer p, with neighborhood $N(p)$, for a target identifier i selects the next peer or set of peers, $R(p, i) \in N(p)$ to forward the message. Due to the nature of small-world networks, decentralized, greedy routing strategy provides the best performance in ring overlays. For a target identifier i, peer p selects the closest preceding link, $d \in N(p)$ to forward the message. Since, there are always k intervals, routing converges in $O(\log_k(N))$ hops.

- Maintenance Strategy: a maintenance strategy is required to maintain the structural integrity while peers go offline or network connection fails. Joining is handled explicitly by all overlays using a join protocol, whereas leaving/failure is implicit, thus requires a maintenance strategy to maintain the connectivity of the underlying graph. As Aberer et al. [26] point out: "The practical usability of an overlay network critically depends on the efficiency of the maintenance strategy." The goal of this work is the enhancement of maintenance, so that these overlays can achieve reversibility.

3.2 Beernet, A Rectified Chord

In our study, we use Beernet [13], which hosts a transactional key-value store, for experimentation. This is a straightforward SON that supports all the maintenance principles. It is an example of the reference architecture (see Section 3.1). Also, it is a non-trivial complex system with interesting global behavior, and a practical scalable transactional store. Beernet has a design similar to that of Chord, the canonical ring-based SON, except that Beernet has correct lock-free join/leave/failure handling protocols. The join/leave handling

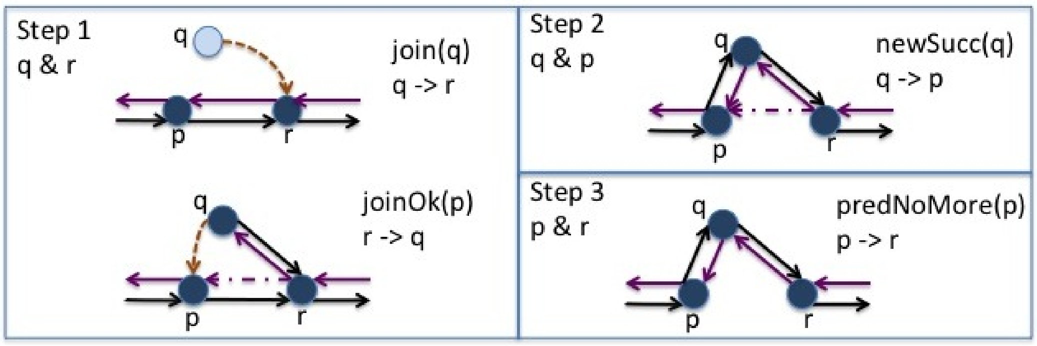


Figure 3.1 – Three steps of Join Algorithm: contact the successor, contact the predecessor and acknowledge the join.

in Chord requires coordination of three peers that is not guaranteed due to non-transitive connectivity (i.e., A can talk to B and B can talk to C $\not\Rightarrow$ A can talk to C) on the Internet. In contrast to Chord, Beernet does not assume transitive connectivity. This makes Beernet more resilient on the Internet-like scenarios.

To avoid locking, the join/leave operation in Beernet is done in multiple steps, where each step is a single message round trip between two nodes. This greatly simplifies the join/leave since it does not have to handle special cases for failure. Figure 3.1 shows the three step join procedure, where a node with virtual id q joins the ring in between nodes with ids p and r, $q \in]p, r]$. A consequence of nodes joining in two steps is that the ring is structured as a core ring surrounded by branches. The branches contain nodes that have done one step of the procedure. As shown in Figure 3.1, after the first step, peer q is on a branch, as it has not yet established a connection with its predecessor and q will continue working on a branch (where r is the root of the branch) if it fails to establish a connection with p. During the last two steps, there are no changes of responsibility, so the address space is already consistent after the first step.

Figure 3.2 shows a general structure of Beernet network, where red (dark in B/W) nodes are organized into a ring, green (gray in B/W) nodes are on branches and blue (light-gray in B/W) nodes are isolated. Ringlets are formed (by k, l, and m in Figure 3.2) due to logical partition of the system triggered by the stress in the operating environment, e.g., high churn (i.e., nodes failing and being replaced by new correct nodes) or physical partition.

In Beernet, unlike Chord, a node always forwards a message to the responsible node in order to take into consideration any branch in between, and if a node considers its successor to be responsible, it sets a flag in the message. Due to branches, the guarantees about proximity offered by Beernet routing correspond to $O(log_k(N)+b)$, where b is the distance to the farthest peer on the branch.

Functionalities of Beernet For Beernet, which hosts a transactional key-value store, we can identify an ordered set of functionalities, $Op_{total} = \{TR, Op_{DHT}, Routing_{log}, Routing_{lin}, NONE\}$.

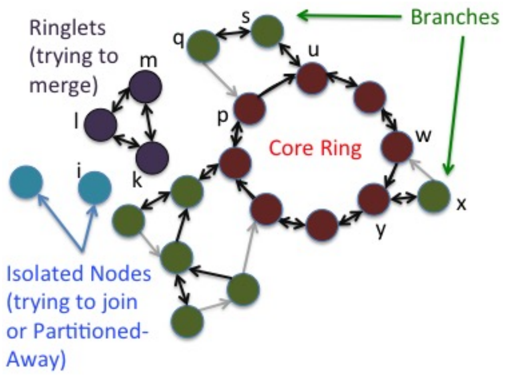

Ringlets (trying to merge)
Branches
Core Ring
Isolated Nodes (trying to join or Partitioned-Away)
q
s
u
m
l
p
w
k
i
x
y

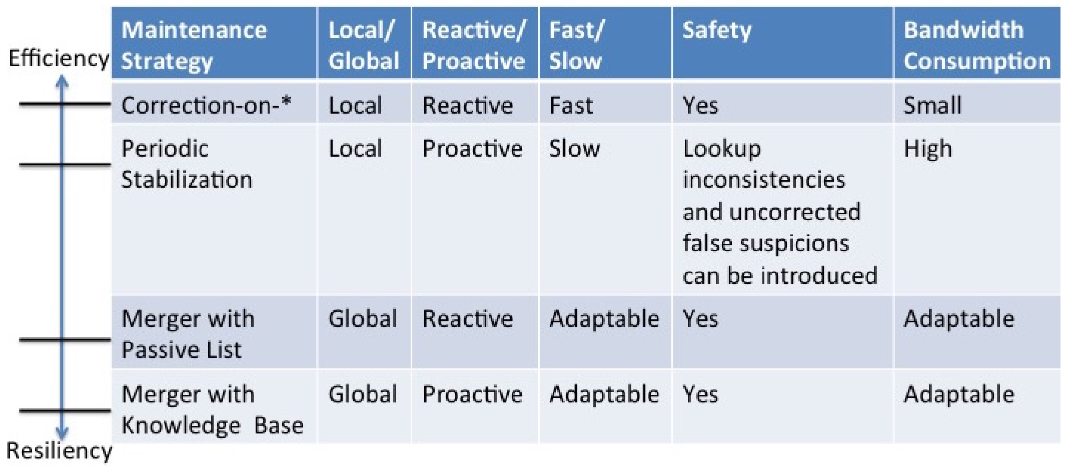
Figure 3.2 – General structure of a relaxed ring

- TR = Transactional functionality.
- Op_{DHT} = Basic Distributed Hash Table (DHT) operations, $get(Key, ?Value)$ (binds $Value$ to the value stored with key Key) and $put(Key, Value)$ (stores the value $Value$ associated with key Key). These operations only access the storage of the node, who is responsible for the corresponding key;
- $Routing_{log}$ = Logarithmic routing, considered to be the efficient routing for ring overlays;
- $Routing_{lin}$ = Basic routing by following the successor pointer of each node, i.e., $O(n)$ routing.
- $NONE$ = No functionality.

In case of our representative system, Beernet, Op_{total} is ordered, i.e., if the system supports transactional functionality (which is the highest-level) this implies that all the lower-level functionalities are also available. Following the definition of reversibility (see Section 2.1), the current set of available operation at a node p, $Op_{avail}[p](t) \subseteq Op_{total}$.

3.3 Maintenance Strategies

The goal we investigate in this thesis is to build reversible systems, for which self-healing is crucial. Self-healing in a SON can be achieved by its maintenance strategy. A Maintenance Strategy maintains the structural integrity of a SON while peers go offline or network connections fail. Chord [11] uses *Periodic Stabilization*, whereas DKS [12] and Beernet [13] rely on *Correction-on-Change/Use*. Then, there are gossip-based strategies, e.g., T-Man [63, 64], which can construct and maintain a SON. So, why not just use any of these strategies? As we have found, these strategies are complementary: Correction-on-change is much more efficient than gossip; whereas gossip is much more resilient. Therefore, we organize the maintenance strategies of overlays using *Efficiency* $\leftrightarrow$ *Resiliency* spectrum,


Efficiency
Resiliency

Maintenance Strategy	Local/ Global	Reactive/ Proactive	Fast/ Slow	Safety	Bandwidth Consumption
Correction-on-*	Local	Reactive	Fast	Yes	Small
Periodic Stabilization	Local	Proactive	Slow	Lookup inconsistencies and uncorrected false suspicions can be introduced	High
Merger with Passive List	Global	Reactive	Adaptable	Yes	Adaptable
Merger with Knowledge Base	Global	Proactive	Adaptable	Yes	Adaptable

Figure 3.3 – *Efficiency* ↔ *Resiliency* Spectrum of Overlay Maintenance Strategies with their properties

as shown in Figure 3.3, where maintenance strategies at the top are efficient, but not resilient as stress increases, whereas the strategies at the bottom are resilient, however lack efficiency. We introduce *Knowledge Base (KB)*, as a simple form of gossip-based strategy, where each peer maintains a best-effort view of the global membership of the system and conduct maintenance using that knowledge. Apart from the extremes, we organize the remaining strategies as per the spectrum: Periodic Stabilization (PS) [11] and ReCircle [37]. PS can be seen as a weak form of gossip, where each node exchanges periodic messages with its successor to maintain its immediate vicinity. ReCircle has two parts: a PS algorithm for regular maintenance and a partition-merger. The partition-merger is a gossip-based maintenance, however as proposed in [37], the merger is triggered in a reactive manner to restrict the number of gossip messages, thus strives to achieve efficiency. Our goal is to get both efficiency and resiliency, so that a SON can achieve reversibility against high-stress. The philosophy behind our work is similar to that used by Plumtree [65].

Apart from this spectrum, the maintenance strategies of overlays can also be classified along two dimensions: local/global and reactive/proactive (proactive is sometimes called "cyclic" or "periodic"). Following [66], we classify the local maintenance strategies (where each node conducts maintenance locally, based on its local knowledge): periodic stabilization as proactive and correction-on-change/use as reactive. The overlay merge algorithm (*Merger*) of ReCircle spreads the maintenance globally. The Passive List (PL) approach [37] (where, each node gossips about the nodes it falsely suspects) triggers the Merger in case of extreme events, thus reactive to the operating environment. In contrast to the PL approach, using KB each node triggers the merger periodically by sampling its local view of the global membership of the system. Thus, the set of strategies we consider covers all points in the two-dimensional space of local/global and reactive/proactive. We describe each principle and its integration with our representative system, Beernet.

3.3.1 Correction-On-Change and Correction-On-Use

These principles were introduced by DKS (we will refer to these principles together as *Correction-on-**). Correction-on-change is concerned about join/leave/failure of nodes. Whenever such events are detected, successor and predecessor lists are updated and the correction of pointers (successor, predecessor and fingers) is triggered. Correction-on-use mainly corrects the fingers. Every time messages are routed, information is piggybacked to correct fingers. As a result more network usage makes the routing table more accurate. Thus, correction-on-use provides self-optimization and self-configuration, whereas partial self-healing is achieved through correction-on-change. These two principles are efficient in terms of bandwidth consumption and rapid response against an event. They update the local state of a node whenever an event like join/leave/failure is detected, which is important to survive an inhospitable environment without introducing any inconsistency. However they fall short when the stress of the operating environment increases beyond a threshold. Without any event, no healing or maintenance is done (i.e., lack of liveness), making these mechanisms a restricted form of self-healing.

Beernet [13], our representative SON, uses both mechanisms for overlay maintenance. As healing is completely dependent on detection of join/fail events, performance of the failure detector plays a crucial role. A QoS-aware eventually perfect failure detection is integrated [67], as described in Chapter 5. When a node suspects another peer, it updates its own successor and predecessor list. If the suspected peer is the successor of the current node (p) then it adjusts its successor pointer by choosing the first one in the successor list (suppose r) and triggers the recovery mechanism by sending a *fix* message to r. When r receives a *fix* message from p, it makes p the new predecessor if its current predecessor is suspected or p is a better predecessor than the current one. The last step of the failure recovery is the *fixOk* message, which is triggered by r, after *fix* is accepted. In this step, p fixes and propagates the successor list. In case of a false suspicion, the failure detector triggers an *alive* message. Now, there are 3 cases to consider; the falsely suspected peer is: i) a better predecessor, in that case the predecessor pointer is corrected and the peer is added to the predecessor list; ii) a better successor, which now requires the correction of the successor pointer and addition to the successor list, also a *fix* message is triggered to run the protocol, this is needed in the case when the old successor also falsely suspects the corresponding peer; iii) any other peer, in that case no special action is required. In all these cases the falsely suspected peer is removed from the crashed list.

3.3.2 Gossip-based Maintenance

In gossip algorithms, information is disseminated is a similar way rumours are spread in social networks. These algorithms are also referred as epidemic algorithms, as they spread information similarly as a virus is spread in biological communities. Demers et al. first used gossip algorithms to maintain a replicated database [68]. When a replica receives any change, it updates its local state based on the change and gossips the change with other replicas. Thus, the replicated database is maintained as updated and consistent. Since then, due to their immense simplicity, scalability, and robustness to failures, gossip algorithms are used to solve various problems.

Gossip-based overlay maintenance strategies are highly resilient against inhospitable environments, but costly in terms of bandwidth consumption and slow to react against events. Using such strategies, each peer maintains a state (local knowledge of the overall system) and uses this knowledge to conduct maintenance. By some rule (e.g., periodically, on demand, etc), each peer communicates with other peers by exchanging the state, and updates it based on the new knowledge that was discovered in the received state from another peer (e.g., if the state is about global membership of the system, then each peer can accumulate a certain sample of the network depending on the size of the state). Furthermore, the way peers decide to contact one another could be either simply random, or driven by the locally accumulated knowledge (e.g., in T-Man [64] peers try to contact not uniform random peers, but those that are known and have closest ID).

3.3.3 Periodic Stabilization

Chord uses this simple idea of a periodic correction mechanism for ring maintenance, where each peer periodically checks the validity of its predecessor/successor pointers. Periodically a peer asks its successor about its predecessor. If it is itself, it does nothing; however if it is a different node, then it is probable that this new node is a better successor for itself. Thus periodic stabilization can be seen as a weak form of gossip, where by exchanging periodic messages with its successor a node attempts to maintain its immediate vicinity. However, this proactive mechanism might yield a slow response when facing an inhospitable environment. As discussed in [69], lookup inconsistencies and uncorrected false suspicions can be created in real implementations. Also, [70] analyzes that inconsistencies can appear in Chord (which uses periodic stabilization for ring maintenance) due to churn. According to [36], for a ratio of churn to stabilization frequency, while doing a lookup the longest finger of any peer is always found to be dead, which degrades routing efficiency. In order to avoid this, it is required to trigger periodic stabilization often, making an inefficient use of bandwidth. Thus, we can say that this proactive local mechanism is complementary to reactive local correction-on-* principles: the correction-on-* require no extra messages in case of no failures, but do not work when no event is detected, whereas periodic stabilization needs no event to execute (i.e., it has liveness), but is costly in terms of bandwidth consumption.

We incorporate periodic stabilization into our representative system. As described above, every δ time units each node p checks for a better successor candidate. To do this, p asks its current successor q about its predecessor pointer and updates if it finds a clockwise closer node to p than q. On the other hand, q also tries to update its predecessor pointer if p is a better candidate than the current one.

3.3.4 ReCircle

ReCircle [37] extends periodic stabilization to react to extreme events like network partitions and merge. It has two parts: periodic stabilization, as described before, and merger. Periodic messages are issued to maintain the local geometry, whereas the merger issues messages that navigate further, triggering awareness and repairing global anomalies, thus ensuring eventual convergence towards one ring. The merger is triggered using a *Passive List (PL)*, where each node maintains a list of suspected nodes and whenever a false-suspicion is detected, merger is triggered using that falsely suspected node. The PL approach to trigger the merger is reactive to the operating environment. Once the overlay converges, the messages issued by the merger die out and ReCircle behaves as a normal periodic maintenance algorithm. Thus, this approach strives to restrict the gossip messages. However, due to such restriction, resiliency is compromised in case of particular scenarios, as our investigation exposes (this will be explained later). Here we provide a brief sketch of the merger mechanism.

Each node maintains a queue, which holds the identifiers of all nodes that need to be fixed, i.e., areas that violate the ring's geometry and introduce inconsistencies. This queue is populated by the passive list. Due to stress, e.g., churn, network partitions, network congestion and so forth, a node might suspect others (some of which are falsely suspected) as being failed. As the stress diminishes, all the false suspicions are eventually detected by a particular node, thus it populates its queue to trigger gossip with those nodes. Every γ time units each node m dequeues the elements from its queue. For each element n, m generates an event called *mlookup(n)*, to fix a possible inconsistent location n on the identifier space. Furthermore, along with *mlookup(n)* by m, n also makes an *mlookup(m)*. Each *mlookup(id)* performs a greedy routing to the problem area defined by id, which is similar to a normal lookup operation. Once it reaches the peer responsible for id, it tries to fix the ring by triggering the same mechanism as periodic stabilization. Also, an *mlookup* tries to fix any wrong successor/predecessor pointers while routing. Further *mlookups* are generated to carry on the repairing of the ring in the clockwise direction. Each *mlookup* spreads this fixing process by generating new *mlookups* from random identifiers on the ring. This is done by enqueuing id into the queue of random nodes selected from the routing table of the current node. This is controlled by a knob, called the *fanout* parameter, to trade-off bandwidth consumption and convergence time. Thus, merger can be made adaptable in term of convergence time.

3.3.5 Knowledge Base

In this thesis, we introduce *Knowledge Base (KB)* as a simple form of gossip-based maintenance strategy. The main idea behind KB addresses two complementary purposes: to provide necessary knowledge for the completion of joining of new peers and to trigger the merger of ReCircle (i.e., the gossip algorithm) in a proactive manner. As the churn intensity increases the frequency of unsuccessful join attempts also increases. The first step of the join protocol of a SON is to do a lookup for a successor and after receiving a response, the new peer becomes part of the SON. With the increased churn, lookup failure rate also increases, resulting in pending joins where new peers keep on waiting to receive responses of their join requests (see Section 6.1). This leads to isolation for the new peers. There are two ways to make such isolated nodes be part of the SON: i) by adding such nodes to the node queues on the overlay, thus triggering the merger of ReCircle in a proactive manner; ii) by providing a valid join reference and re-triggering the join request. As isolated nodes are still not part of the SON, the nodes on the overlay have no reference to these nodes, thus no healing mechanism will be effective. In order to apply any of these proposed solutions the knowledge about the alive peers on the overlay is required. Also, triggering the merger only in a reactive manner is not sufficient to provide complete self-healing in an inhospitable environment. As the churn intensity increases, the overlay might be partitioned in such way that there is no physical communication problems among the partitions (i.e., logical partition of the overlay network). This might also happen as well due to network partitioning (discussed in detail in Chapter 7). For such scenarios, no suspicion event gets issued. As a result, the passive list mechanism used in the ring merge algorithm [37] fails to trigger the merging.

When there is simultaneous network partition and churn, the number of strangers (refer to Chapter 7) increases due to the interaction between these two stress sources, merger with passive list fails to achieve reversibility (refer to Chapter 7). In order to ensure reversibility up to the best limit, for such scenarios and initiate merging of overlays in the same partition, require more knowledge than a passive list to trigger the merger, which calls for building of a *Knowledge Base (KB)* at each peer. The *Knowledge Base* at each node is the best-effort view (complete/partial) of the global membership of the system. It extends ReCircle [37] in two ways: i) consider bigger set of nodes than the passive list, ii) proactive triggering of merger: even if there is no false suspicion (explained shortly). The nodes in the KB are not monitored by the failure detector at a peer, thus causes no change of the embedded small-world graph of the overlay. The knowledge base at each node can be accessed through an API. There can be different levels of knowledge base:

- Passive KB: Such KB can be built through listening only. At peer p, for each node $a \notin KB_p$ that p comes across (while routing or as a member of its current neighborhood), the virtual identifier and the network reference of a is added to KB_p. Building of knowledge base at each node is based on passive observation; the list built by this strategy is never shared with other peers, thus creates no impact on bandwidth consumption or scalability of the overlay. However, such KB may be out of date quicker.

- Active KB: Using this strategy a node communicates with others to enhance its KB. This can be a weak algorithm, such as each joining node just informing others of its existence, or a stronger algorithm, such as gossip where each node asks a random node to send its KB that it unions with its own KB. The result is that the KB converges “faster" to maximum information, so that when a partition arrives, it is in the best possible condition for surviving (least possible strangers). The effect of the active approach is mainly on the convergence time of KB. However this approach causes extra load on available bandwidth.

- Oracle: This is information coming from “outside the system". In order to use the first of two alternative solutions proposed above to deal with isolated nodes, an oracle is required. Also, as the duration of the underlying network partition increases, more and more nodes in each partition are replaced by new nodes due to churn. Thus, eventually the nodes in one partition will become complete strangers for the nodes on other partitions, as $(\bigcup_{p_i \in P_1} KB_{p_i}) \bigcap P_2 = \emptyset$ (and vice versa), here P_1 and P_2

are the sets of peers of two partitions. This may also happen for short/arbitrary duration of partition depending on the intensity of churn. In Chapter 8, we discuss the stranger syndrome using an analytical model. For such scenarios, the knowledge base built at each node falls short to merge the overlays as the partition ceases. As two overlays have no knowledge about each other, the system requires the intervention of a third-party to gain reversibility. An oracle can be part of the API, using which the application layer or a bootstrap server can give information to the system.

In order to use the KB efficiently, it is necessary to define a proactive sampling technique that periodically chooses elements of the KB. Since the KB can be very large, it is not practical to use all elements of the KB. Every σ time unit, each node randomly picks up an element e from its KB, where e is neither member of its current neighborhood nor is currently suspected, and enqueues e into its queue. Here, σ is the mean of a Poisson distribution of times when a periodic sampling happens and is a tunable parameter, which affects the convergence time of the overlay merging. This mechanism can be made more efficient by making σ adapt to the operating environment, which we leave as future work.

The KB at each node is always growing. It may or may not converge to cover all nodes in some partitions, depending on the operating conditions. But the number of “dead" node references in the KB will always grow, which introduces a garbage collection problem. We can add a time-stamp to all references in KB and optimistically remove all “sufficiently old" nodes from the KB, implementation and validation of this is left as future work. The knowledge base presented in our work is a generalization of the passive list proposed in [37], which only keeps track of currently suspected nodes. As a node only monitors the members of its current neighborhood, so can only trace failures in this set. So, we can say that the knowledge base is a super-set of the passive lists at any time, thus enriches the nodes with required knowledge to trigger overlay merge even for long-term partition.

Chapter 4

Stress and Behavior

We want distributed applications to survive in stressful environments, where there are continuous faults and high numbers of concurrent failures. For this purpose, in this thesis we investigate reversibility for practical distributed systems and applications in stressful operating environments. As a prerequisite of which, it is necessary to identify the scenarios which cause the operating environment of a distributed system to become stressed. In this chapter we identify the main stress sources. Finally, this chapter concludes by identifying a set of high-level properties to capture the behavior of a ring-based SON.

4.1 Sources of Stress

The goal of this section is to identify the sources of stress, which cause the operating environment of a distributed system inhospitable. We define an inhospitable operating environment for a system in which certain stress parameters can potentially reach high values and temporarily increase without bound. In other words, we need to organize the global operating space of distributed systems in term of non-malicious failure scenarios. However, the operating space of a distributed system, to be specific a *Structured Overlay Network (SON)* is large: given n nodes connected by communication links, any set of nodes may crash, any communication link or set of links may slow down or fail, permutation of these failure scenarios can lead to infinite possibilities. In this work, we consider and structure all such possible failure conditions using five dimensions, i.e., five stress sources. The rest of this section will define each of these stress elements and how we have quantified or captured them in our fault model.

4.1.1 Churn

Churn is the most usual and basic scenario that a SON faces during its lifetime. The term churn is used to express the measurement of peers joining or leaving the network or failing during a given period of time. Churn causes the operating environment of a system to be stressful, as due to churn the state of the system gets perturbed. That is why handling high churn or bursts of churn is challenging for any distributed system.

The literature of SON mostly addresses churn as the only failure scenario in order to establish the basic fault tolerance and self-management characteristics. However, in most cases churn remains under a certain limit. Studies such as [71, 72] show that a peer-to-peer environment experiences high churn rates, where nodes continuously join and leave the system and their up-time in the overlay is short. Several studies [73], [74] have been conducted to understand peer behavior or churn in different peer-to-peer systems. As per [73], the majority of peers are long-lived, however the remaining short-lived peers join and leave the system at high rate, which comprises a large portion of sessions. According to this, we can say that even systems with low/average churn face high peaks. So, it is to worth investigate the reversibility and behavior of a SON against excessive churn or bursts of churn.

We present the definition of churn, used in this work. As nodes join and leave/crash during churn, with equal probability for join and leave events, we can say that a node changes its identity during churn with only two events. In other terms, if we assume equal probability of join/leave event and a single event per time unit, then every two time units, a node will leave and a new node will join the network, i.e., every two time units the total number of nodes on the network will be same, whereas only a single node has a changed identity. We assume that churn varies over time, and that the average number of correct nodes at any instant is constant. We have defined churn as the percentage of nodes turnover per time unit.

4.1.2 Network Partition

Network partition is one of the worst scenarios that a SON might face during its lifetime. Here, we consider only the partitioning of the system due to partitions at the physical network, i.e., failure(s) of underlying physical link(s) causes the nodes of a SON to be divided into multiple disjoint sets during the partition, where a node can communicate with the nodes of its own set, but is unable to contact the nodes in the other sets. Any long-running large distributed system is bound to come across network partitions during its execution. Several reasons may cause the underlying network to partition: link failure, router issues (failure, misconfiguration, overloading), malicious activities (denial of service attacks), software bugs or physically damaged network equipment [75, 76, 77, 78]. For example, as a consequence of several natural disasters, it was exposed that the global network connectivity depends on a few active “choke points" [79]. Failure of such choke points under natural calamity or similar events can lead to network partitions [75]. Apart from these, policy based issues or conflicts can also cause inaccessibility across regions, resulting in network partitions [80, 81, 82]. Though partitioning can be seen as massive churn, it creates distinguishable operating conditions for a SON by itself, that’s why it deserves to be a separate dimension in our proposed global operating space. The only parameter, which characterizes this dimension, is the number of partitions, i.e., the number of disconnected sub-graphs in the system.

4.1.3 Network Dynamicity

Network dynamicity happens when communication links slow down, resulting in congestion and packet loss in the network. The impact of such scenarios cause false suspicions in the system. This poses challenges, as each false suspicion event triggers failure recovery, routing table update and other mechanisms, thus more traffic for the underlying physical network, aggravating the congestion or packet-loss. Huge numbers of frequent false suspicions create a challenging operating condition for a SON, in the face of which, the SON struggles to maintain its structure. We define congestion in our fault model, as the percentage of node slow-down by t_{slow} time unit per window of $T_{congestion}$ time units. We define "a node slow-down by t_{slow} time unit", when each outgoing message of the node experiences an additional delay of t_{slow} time unit.

Packet-loss in our fault model is assumed following the drop-tail loss model as explained in [83]. We assume one message of our system is equivalent to one packet, which is reasonable as the message size is not so large. Each node p is assigned a maximum Bandwidth, $BW(p)$, in terms of number of messages per time unit. During each time unit, messages are considered sequentially. For each message with a sequence number of n during a time unit, if $n < BW(p)$, the message is lost with a probability of $\log_{BW(p)}(n)$, otherwise, the message is lost with probability 1.

4.1.4 Workload

The application workload often proves to be a key factor in determining the hostility of the operating environment. A particular amount of workload may work quite well for a particular SON, whereas might make another limited resource SON unresponsive. Workload beyond the physical capacity of the system will overstress it. The workload can be defined as the number of transactions per time-unit for a SON with transactional DHT, along this dimension.

4.1.5 Ring Size

The last dimension of the proposed operating space of a structured overlay network is the size of the network, the number of nodes in the system. This dimension in fact evaluates the SON's scalability.

4.1.6 Stress Elements Explored

In this thesis, we have followed an incremental approach to investigate reversibility of our representative system against several stress elements. We explore one stress source, with zero or constant value of other stress parameters. This thesis presents our investigation of *three* stress elements. Below we enlist the stress elements addressed in this thesis and we rationale for the choice of these stress elements.

- Churn: As already mentioned, churn perturbs the state of a system. Extremely high churn disturbs the global state of the system at a high rate, i.e., before recovering from one churn event by doing corresponding corrections of the global state, the system faces another change in the state due to the next churn event. Here, the churn events we consider are the joining or failure of nodes. We have left out gentle leave (i.e., voluntary leave) in our work. We consider gentle leave as a special case of failure. Also, when a node crashes, this event creates more stress on the system than the scenario, when a node disconnects voluntarily and node-crash corresponds to partial failure in the system. Furthermore, churn or burst of high churn is the usual scenario that a long-running distributed system may expect frequently during it lifetime. So, reversibility against extremely high churn is worth to be investigated.

- Network partitioning is another example of partial failure in a distributed system, however in this case failure of communication. Partition of the physical network also can be of short or long duration. This poses two different types of challenges for a system: (i) execution during the partition and (ii) execution at partition repair (i.e., when the partition ceases). Due to such distinguishable operating condition, reversibility against network partitioning deserves to be investigated.

- Network dynamicity impersonates partial failures in a distributed system. Due to congestion and packet loss, false suspicions are issued. As a result of which nodes which are falsely suspecting others initiate the corresponding state changes and when the false suspicions are detected they must revert those changes and retain the original state. Thus, these stress elements cause unnecessary resource consumptions and pose a risk for the system to reach an inconsistent state. We also investigate these scenarios in this thesis.

Apart from the investigations of these stress elements individually, we have explored interaction between two stress elements. In Chapter 8 we explore the scenarios where there exists simultaneous network partitioning and churn.

4.2 Behavior of a SON

We define the term “behavior”, with respect to a SON using a set of metrics, which capture the behavior of a particular SON at various levels. In the literature, various metrics are proposed to analyze the performance of a SON from different perspectives. In this thesis, we have chosen the metrics that capture the impact of a particular environment on the operation of a SON, and can be easily measured. For a systematic and thorough investigation of a SON with transactional DHT, we have identified three levels. Below we present the parameters at each level.

Data Level Parameters Percentage of failed transactions, percentage of lost keys, percentage of inconsistent replicas, percentage of lost updates.

The data level parameters signify the impact of a particular environment on the DHT and data level operations. The last parameter, percentage of lost updates provides useful information for applications running on top of a SON regarding the ratio of successful updates of the stored data in a particular environment. This parameter is mostly dominated by percentage of failed transactions, however other scenarios may also contribute. Suppose, the replica set of key k_i is formed by peers a, b, c, d, where a, c, d has the latest value with version 7 (i.e., there were total 7 updates) for k_i and b holds the old value with version 5. Now, if a, c, d leave or crash or become unavailable (for example, due to partition), then the old value stored in b will be replicated to the new responsible for key k_i. In this scenario though there was no failed transaction, two updates are lost. Another scenario: suppose transaction is designed for an application in such way that the update operation of a key depends on its old value, in this case if the key is lost, then it doesn't invoke any update, contributing to the lost updates. All other data level parameters are apparent from their titles.

Connection Level Parameters Number of times imperfections introduced, percentage of time a node experiences imperfections, percentage of nodes on core ring.

In all ring-based SONs, each peer keeps a successor and a predecessor pointer to maintain the structure. The connection level parameters assess the deviation of the ideal ring structure, i.e., the embedded graph, during adjustment with a particular environment. The parameter *Number of Imperfections Introduced*, counts the total number of times peers falsely suspect their ideal successor or predecessor pointer and choose an imperfect peer. The next parameter, *percentage of times a node experiences imperfections* accounts for how long a peer goes through such imperfection. These two parameters reflect the impact of false suspicions on the ring. The parameter *percentage of nodes on core ring*, tries to measure the rigidity of the system. In other terms, it finds out the maximal ring in the system and reports percentage of nodes on it.

Routing Level Parameter Number of messages generated.

The routing level parameter, *Number of messages generated* shows how many messages are generated to cope with the changed environment. As already mentioned, these metrics are general enough to be applied for any SON with transactional DHT.

Chapter 5

Failure Detection and Replica Management

As prerequisites of our study, we have enhanced some aspects of our representative *Structured Overlay Network (SON)*, Beernet. As part of these efforts, we present a QoS-aware self-adaptable failure detection algorithm [67]. Also, we have augmented replica management of Beernet with an optimistic, low cost data migration protocol as part of failure recovery algorithm, in order to achieve improved availability and consistency of replica sets [67].

5.1 Failure Detector

Partial failures are frequent events in distributed systems, with which they must cope with. For efficient (i.e., reactive) managements strategies, e.g., Correction-on-Change, *Failure Detectors* play an important role in a system's operation, because most of these changes (nodes crashing or leaving the network) are detected and notified by the failure detection module of each peer. Perfect failure detection is impossible especially for Internet style networks. An *Eventually Perfect Failure Detector* is the best that can be achieved on the Internet. Along with Strong Completeness, an eventually perfect failure detector requires *Eventual Strong Accuracy*, which implies that any false suspicion of failure will be eventually corrected. This is feasible to implement because, if the unreachability, thus failure suspicion, was caused by problems of the underlying communication link (failure or slow), it will eventually go away as the link recovers, thus be able to communicate with the suspected peer. This is the kind of failure detection that a system's reactive algorithms rely upon, however, *Quality of Service (QoS)* of failure detection modules is crucial for ensuring good system's performance, as frequent false suspicions may cause instability in the SON. Also, longer periods to detect a crash or correct a false suspicion will allow greater inconsistency in the system. For these reasons, maintaining QoS of failure detection is of utmost importance. This section presents an implementation of a self-adaptable eventually perfect failure detector, which adapts with the environment, providing QoS.

The existing implementation of eventually perfect failure detection in Beernet [84] is a generic one, guided by that in [85]. The idea is very simple: each peer p periodically sends a *ping* message to all other peers in Π, where Π is the set of peers p knows about. After the ping message is sent, p launches a timer, which corresponds to the timeout for responses from all peers in Π. Each node q, which receives a ping message from p, immediately responds with a *pong* message. When p receives a pong message from q, q is added to *alive* set. When the timeout is triggered, all the peers, which are not in the alive set are suspected to have failed. Then p starts a new round and repeats the same way of collecting responses from other peers. If a pong message is received from a peer r, where $r \in suspected$, implies that r was falsely suspected in an earlier round. This is detected at the next timeout, when r belongs to both *alive* and *suspected* sets. In this case the false suspicion regarding r is corrected and the timeout is increased by a fixed value, Δ in order to prevent such false suspicions.

The algorithm above has two problems. The first issue is, the timeout is global for all connections of a node whereas the round trip time is very often different for each connection. For this global timeout, the detection process of a peer is driven by the speed of the slowest connection. Let us consider the function of a failure detector at a particular peer a, where a monitors b and c. The connection between a and b is very fast but between a and c is very slow. In the first round, a will receive response from b within due time but will falsely suspect c. Later, when a will receive response from c, it will increase its timeout, this way a will continue increasing its timeout period until there is no false suspicion, meaning that the timeout is adapted to the slowest connection. The second problem with the above algorithm is that it only increases the timeout to adapt to the round trip time (RTT) between two nodes, whereas the RTT may vary wildly along time, in a long running distributed system. Due to temporary congestion, RTT may become extremely high, leading to false suspicion or increase of timeout and after some time when the congestion goes away it may drop to a more regular value. Thus, it is necessary to lower the timeout period to adapt to the RTT, without creating any oscillation.

The expectations from the failure detector module are threefold: to reduce the number of false suspicions, to correct mistakes quickly, and also to detect failures quicker. Unfortunately, these goals are contradictory: the time to detect failures or correct a mistake depends on the timeout period. Lower timeout delays lead to quicker detection of failures or correction of mistakes, but in order to reduce the number of false suspicions the timeout delay needs to be sufficiently large to cope with slight changes in the environment (i.e., noise). In order to determine the perfect or best value of timeout period for each connection adapting to the RTT of each connection is the only option.

Based on the above observation, we present self-adapting failure detection, a failure detection implementation that self adapts to take into consideration the latency of each individual connection monitored by it. Additionally, it also keeps a safety period to accommodate the variability in a network connection. Each node maintains a history of the last k RTT measured for each connection and the timeout period of next round for each connection is determined based on this history. We will explain this using an example of a pair of nodes (as for each connection the procedure is the same), where a is monitoring b (i.e. connection $a \leftrightarrow b$). At each round, a sends a *ping* message to b, each *ping* message is time-stamped. When b receives a *ping*, it replies with a *pong*, including the time-stamp of the corresponding *ping*. As soon as a *pong* message from b is received, a calculates the RTT of the connection between a and b for this round. After the timeout, if no *pong* message from b is received, a starts suspecting b, if b is not already in the *suspected* set. However, if b is in the *suspected* set and a *pong* is received from b in the current round, then the false suspicion regarding b is corrected. After deciding the status of b for the current round, a starts preparing for the next round for monitoring the connection towards b by calculating the timeout period.

The timeout Period for the next round is captured using the average RTT over the last k rounds plus $m \times$ Standard Deviation of RTT over the last k rounds.

The first part of the above equation is used to estimate the expected value of RTT for the next round. However, due to unpredictability of a large and complex network like the Internet, a safety period is needed so that in case of sudden changes of environment, the timeout period can cope up quickly. For this purpose, a weighted standard deviation over the history of RTT is the best candidate, where the weight m depends on the variability in the network. Together, as a summation of both quantities, it enables to determine the most suitable value of timeout period for the next round and also auto-adapts with the changing environment. This failure detection algorithm can be expressed as a function of k and m, and the best values for these parameters can be determined for a particular environment, based on experiments, as shown in Section 5.1.2. For the initial rounds, when there is not enough RTT data collected, a predefined timeout period is used. The algorithm employed by the self-adapting failure detection module is shown in Algorithm 1.

5.1.1 QoS Metrics for Eventually Perfect Failure Detection

As already mentioned, QoS of the failure detection module plays an important role in a system's performance and resilience against a particular environment. The expectations from failure detection module are threefold: quick detection of failures, low number of false positives or mistakes, and fast correction of false positives. From a direct mapping of these expectations, we define three primary QoS metrics for an eventually perfect failure detector, which are in-line with those proposed by Chen et al. [86]:

- Detection Time: Time to detect a failure or crash. This is the average duration after which all peers permanently suspects the crashed node.
- Accuracy: Percentage of suspicions which are correct, i.e., percentage of suspicions when the suspected node has actually crashed or left the system.
- Reaction Time/Mistake Duration: Time required to correct a false suspicion. This is the average duration after which a peer becomes aware of a false suspicion and corrects its mistake.

5.1.2 Evaluation

The objective of this section are threefold: i) To evaluate the failure detection algorithm using the QoS metrics as described in Section 5.1.1, ii) To understand the impact of the two

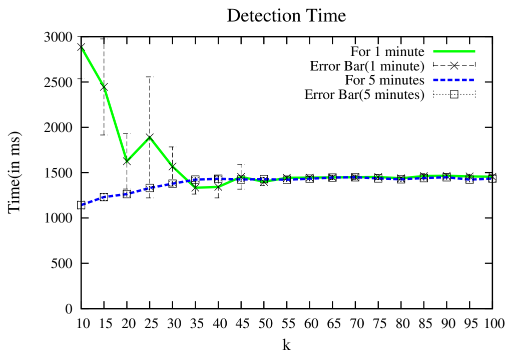
Algorithm 1 Self-adapting Failure Detector

```
upon event ⟨ monitor | p ⟩ do
  MonitoredPeers := MonitoredPeers ∪ {p}
  p.waitPeriod := $INIT\_DELAY$
  send ⟨ ping | self, $timestamp_{ping}$ ⟩ to p
  starttimer(p.waitPeriod)
  end

upon event ⟨ timeout | p ⟩ do
  if p ∈ alive and p ∈ suspected then
      suspected := suspected \ p
      trigger ⟨alive| p⟩
  if p ∉ alive and p ∉ suspected then
      suspected := suspected ∪ p
      trigger ⟨suspect| p⟩
  alive := alive \ p
  expectedRTT := Average of the last k values in p.RTTHistory
  stdRTT := Standard Deviation over the last k values in p.RTTHistory
  p.waitPeriod := expectedRTT + m*stdRTT
  send ⟨ ping | self, $timestamp_{ping}$ ⟩ to p
  starttimer(p.waitPeriod)
  end

upon event ⟨ ping | q, $timestamp_{ping}$ ⟩ do
  send ⟨ pong | self, $timestamp_{ping}$ ⟩ to q
  end

upon event ⟨ pong | p, $timestamp_{ping}$ ⟩ do
  RTT = CurrentTime - $timestamp_{ping}$
  p.RTTHistory := p.RTTHistory ∪ RTT
  alive := alive ∪ p
  end
```



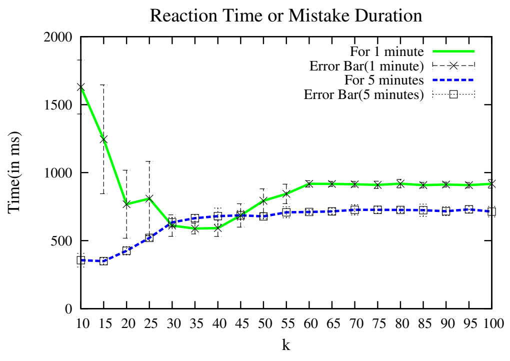


Figure 5.1 – Detection and Reaction Time for various values of k

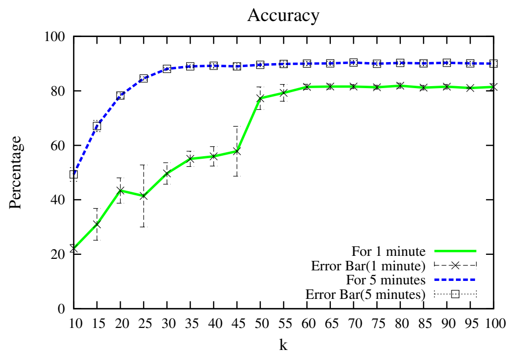


Figure 5.2 – Accuracy for various values of k

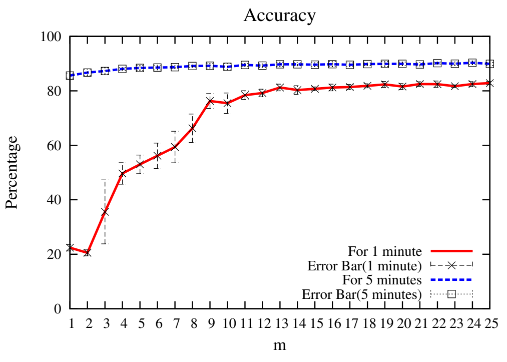


Figure 5.3 – Accuracy for various values of m

parameters (k and m) on the QoS of the failure detection in a given environment, and iii) To tune these two parameters for upcoming experiments. The ideal values of these parameters to achieve the best performance is highly dependent on the environment. It is possible to make the failure detector self-tuning, but we leave this as future work.

To evaluate the QoS of failure detection, a network of 100 peers is used. Correction-on-* principles are used as the sole maintenance strategy. To simulate the underlying network, the end-to-end delays are set based on the empirical distribution of the minimum RTT provided in [87]. During the steady state of the SON, 5% churn is injected. The churn events are modeled as a *Homogeneous Poisson Process (HPP)* with λ_1 events per second, where $\lambda_1 = 10$ for 5% churn on a network of 100 peers. Simultaneously, in order to simulate variability in the underlying network, 5% links (in this work link always refers to end-to-end connection of the overlay) changes connectivity every 5-second, based on the empirical distribution of standard deviation in per-connection RTTs as measured in [87]. The connectivity change events are modeled as a HPP with arrival rate of λ_2 events per second, $\lambda_2 = \frac{5*L}{100}$ and L is the total number of connections (for 100 peers, $L = \binom{100}{2}$) in the system. To evaluate the adapting capability of the failure detection scheme, we have taken measurements for two different durations, 1 minute and 5 minutes.

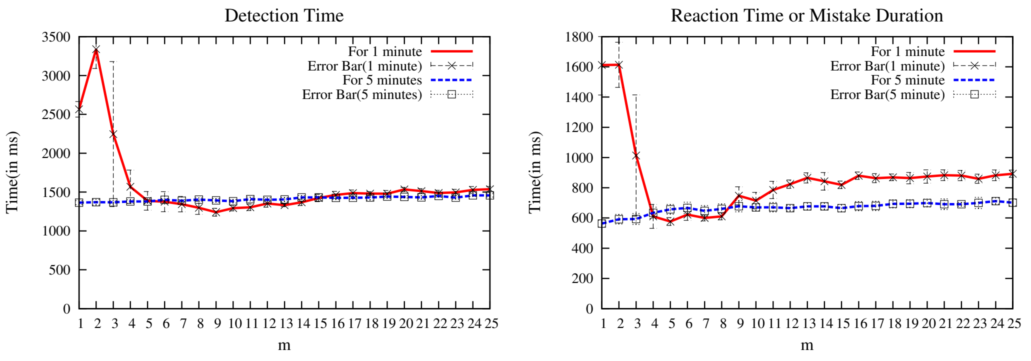


Figure 5.4 – Detection and Reaction Time for various values of m

For this given environment, we have measured the QoS parameters for various values of k, by keeping m fixed to 4. The result obtained is shown in Figure 5.1 and Figure 5.2. The accuracy increases with the buffer size until it reaches a maximum point and becomes stable both for short and long duration. As the environment is changed consistently throughout the experiment, longer history provides more statistical information, however in case of a temporary fluctuation in the environment, storing very long history might have a negative impact on QoS, also buffer size becomes an issue. As it is apparent, after adapting to the environment for longer time, accuracy has improved for the same buffer size. In terms of detection and reaction time, we get good results for buffer sizes of 30 to 40 entries, beyond that we again get an increasing trend for short duration. However for longer experiments, the outcome increases till buffer size of 30-35 as accuracy goes up, after that it remains steady as it already has a good estimate of the RTT and settles for the detection and reaction time for the best accuracy. For the following experiments, the buffer size is set as 30, as this appears to be a sufficient history size for the failure detector to reach the optimal state.

The same environment is used to test the impact of m on QoS of failure detection; Figure 5.3 and Figure 5.4 illustrate the observed results. As expected, accuracy increases with increased timeout period, i.e., with increment of m for short experiment. However, for longer period, failure detector's adapting capability is prominent, as the average RTT dominates the timeout period, i.e., accurately measures the expected RTT which leads to maximal accuracy, thus, making the impact of m fades away. This happens because network variability is modeled as an HPP, however for an unpredictable environment like the Internet, m will have its impact even for longer up time. The detection and reaction time decreases sharply for $m > 3$, remains almost same for 4 till 9 and again shows an increasing trend for short experiment duration. Based on this analysis, we have kept $m = 4$ for the remaining experiments in this work in order to cope up with the environment right from the beginning.

5.1.3 Related Work

There are two implementation strategies for timeout-based failure detectors [88, 89]: i) Heartbeat based, where each process periodically sends "I am alive" messages to all other processes; ii) Ping based, a process monitors another process by sending a ping message and asking for acknowledgement. The ping strategy provides finer-grained control, while also following the design philosophy of Beernet, each node has a set of other peers (where the set is different for different peers), which it monitors, and this matches the ping strategy. However, existing work on QoS of failure detectors is mostly based on heartbeat-based implementation, the reason as mentioned in [88], is the quality of the estimation of the timeout period. In contrast to the heartbeat strategy, in ping strategy the number of variables is twice (for example, transmission, reaction delays). To maintain QoS, these approaches [88, 86] use sampled arrival times to compute an estimation of the arrival time of the next heartbeat. The timeout is set according to this estimation plus a safety margin (which is constant in [86], whereas [88] computes it by using Jacobson's algorithm [90]). Very few works are found on the ping based failure detectors. A message efficient algorithm is provided in [91], but in this approach each process monitors only a single process. In [92], the failure detection is lazy, where the status of the monitored process can be known by making a query to the failure detection module. A process is suspected when the elapsed time for the unacknowledged message is more than the maximum round trip time till observed. However, the maximum round trip time can be much higher than the average response time due to a spike, so this may increase the detection time. The protocol provided in [93] assumes a model of finite average response time; timeout increases logarithmically with the total number of slow messages (triggering false suspicions) and linearly with the number of fast messages (acknowledgements received before timeout) since the last slow message. Whenever, a slow message arrives, the number of fast messages is set to zero and therefore results in a drop of the timeout (if the number of fast messages was greater than zero). This may cause increased false suspicions as the timeout decreases after a false suspicion. Also this is a pull mechanism to get the status of a monitored process, whereas Beernet needs failure detector to push notifications regarding any event.

5.2 Transactional DHT

Our representative system, Beernet, hosts a transactional Distributed Hash Table (DHT) with replication. The performance or correctness of the applications running on top of Beernet using its Transactional DHT service is dependent on the assurance provided by the data layer operations of Beernet. This section provides a brief description of data layer support in Beernet, along with an augmentation of replica management in order to achieve improved availability and consistency among the replicas of a particular data item.

There are three protocols implemented in the transactional layer of Beernet: Two-phase commit, Paxos Consensus Algorithm [94], and Eager Paxos (an extension of Paxos consensus algorithm with an eager locking mechanism, see Section 11.3.1 for a brief description of this protocol), in order to satisfy the requirements of various types of application. Though Two-Phase commit is the most popular approach used by traditional databases, it has two stringent requirements: survival of the transaction leader and all replicas must be updated when the transaction completes. These two requirements are hard to be satisfied in a dynamic environment. However, Paxos consensus protocol relaxes these requirements by using replicated transaction managers and quorum-based commits, thus providing fault tolerance without sacrificing strong consistency. So, in this work, mostly the Paxos consensus protocol is used, as the obtained results can also be applied to Eager Paxos.

Symmetric Replication [70] is implemented in Beernet as a replication strategy. The replica management layer of Beernet strives to maintain a consistent set of replicas under any extreme environment. Let us analyze the operations of this layer in detail under churn. When a new peer joins, it replaces its successor as a member of the replica sets of a certain amount of items. During the join procedure, the successor pushes all such data-items to the new peer that it has become responsible for. Taking values only from a single replica is fine in this case, as if the successor has stale value of a data-item, this will replace a bad replica with another bad replica. However, when there is a failure, it is more important to read from the majority during the recovery, as there is no way for the recovery node to know whether the dead peer was up-to-date or not. The existing implementation of replica management in Beernet only handles the join events [84]. We extend the replica management layer to handle the failures as well. In order to read from the majority during failure recovery, doing transaction for each data-item becomes too expensive and false suspicions make the situation more complex to address. In this work, we have augmented the replica management layer of Beernet, by proposing an inexpensive lazy-migration protocol during failure recovery, which can an achieve eventually consistent replica set for a data-item, without creating any conflict with simultaneous transactions of that particular data-item.

5.2.1 Lazy Data Migration

Each data-item has a version number associated with it, which increases monotonically by each update. The *read* operation during a transaction reads from the majority and returns the value with the highest version number. While committing a write operation, a replica votes for commit if: $version_{new} >= version_{existing}$ for an existing data-item or the data-item is absent. The monotonicity of version number, along with the advantages of symmetric replication is exploited in the lazy migration protocol. Suppose, peer q has successor r and predecessor p, when p suspects q, it initiates failure recovery, now if q is also suspected by r, r takes over the responsibility for all the data items that q was responsible. Due to the symmetric replication strategy, r can find out other members of the replica set of the corresponding data items using the symmetric function, so there is no need to add any expensive group management to the replica sets. Though, there is a replica set per data-item, due to symmetric replication, many replica sets overlaps, which facilitates the data migration. Suppose, the replica set is formed by a, e and m (for replication factor 4 and the $4th$ member q is suspected). Then, r sends a pull request to all of them to do a data migration for the data-items that q was responsible for. If a peer receives a pull request, it retrieves all the data-items belonging to the specified range from its own storage and sends those data-items to the mentioned destination. When r receives the data from any of them, it only do an update of a particular data-item if: $version_{received} > version_{existing}$ for an existing unlocked data-item or the data-item is absent in r. Thus, r may have stale value for a data-item temporarily, if it receives from e (which has old value) before a or m (holding up-to-date value), however eventually r will be consistent for the particular data-item. However, this is not a perfect solution to achieve consistent and complete replica set for each data-item, as there might be non-overlapping replica sets, but this is a trade-off between cost and consistency.

Let us now analyze the lazy migration and simultaneous transactions of a data-item, k_i. Suppose, a, e and m are the replicas for k_i and after q is suspected r has become the member of the replica set. If it is a read transaction and at least one member of the current replica set has the up-to-date value of k_i, then the value with the highest version number will be returned, maintaining strong consistency guarantees. However, if the suspected member of the replica set is the only one holding the up-to-date value of k_i, then there will be update-loss, which is unavailable. For a write transaction involving k_i there can be three possible scenarios, when lock is requested for k_i by the transaction manager:

- r does not have k_i: r will vote for commit, so no inconsistency is introduced.
- r has a stale value of k_i: due to monotonicity of the version number, there will be no inconsistency and r will vote for commit.
- r has an up-to-date value of k_i: this is the ideal scenario, resulting in regular transaction.

So, in any case, lazy-migration of a data-item to the new member of the replica set does not create any conflict with simultaneous transaction of the same data-item. This is a low cost optimistic protocol to achieve completeness and higher consistency of replica sets.

5.2.2 Evaluation

This section evaluates the impact of Lazy-Data-Migration on the data level parameters, as described in Chapter 4. For this experimental study, a network of 100 peers has been bootstrapped. Correction-on-* principles are used as the sole maintenance strategy. As in Section 5.1.2, the statistical properties in [87] are used to simulate the underlying network. A consistent workload is created by injecting transactions, whereas transactions are modeled as a *Homogeneous Poisson Process (HPP)* with $\lambda_1 = 1$ per second. The workload is kept in such way that without churn the data-layer operates perfectly, i.e., no failed transactions and inconsistencies. A transaction is designed as: it reads k_i and updates k_j, here update means k_j is read and if it exists the value of k_j is incremented by 1. The replication factor is kept as 4. As in Section 5.1.2 churn events (join and crash with equal probability) are modeled as a HPP with rate λ_2 events per second, whereas $\lambda_2 = 2 * C$, for a churn of $C\%$ on a network of 100 peers. In order to understand the impact of Lazy-Data-Migration protocol, all data level parameters are measured once with lazy-migration and once without lazy-migration for each value of churn. The two sets of result for each parameter are plotted together so that they can be easily compared. Figure 5.5-5.8 show four data-level parameters for the two sets of result.

As we can see in Figure 5.5, increasing churn leads to more failure of data layer operations. However, lazy migration doesn't create any conflict with transactions, on the other side shows improvement as the availability and consistency of replicas are improved. As

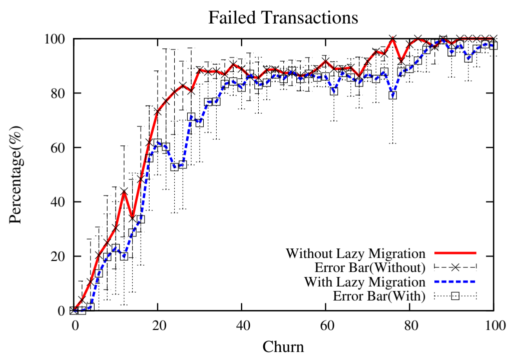


Figure 5.5 – Data Level Parameter: Percentage of failed transactions

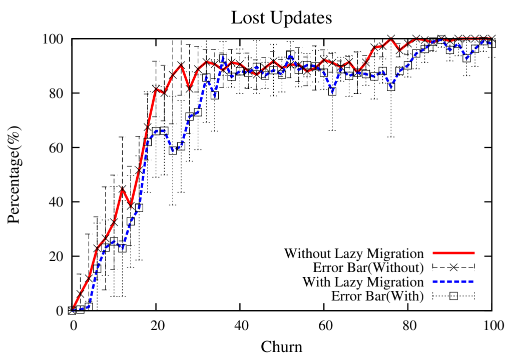


Figure 5.6 – Data Level Parameter: Percentage of lost updates

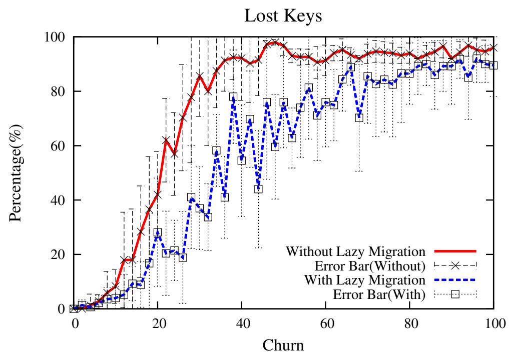


Figure 5.7 – Data Level Parameter: Percentage of lost keys

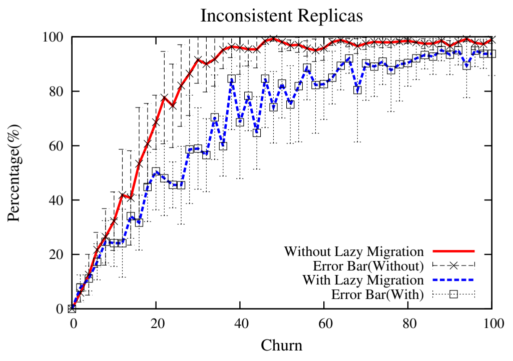


Figure 5.8 – Data Level Parameter: Percentage of inconsistent replicas

expected, the main impact of lazy migration is clearly visible in Figure 5.7 and Figure 5.8, as the integration of this simple, low-cost and optimistic approach leads to less key loss and inconsistent replicas. Figure 5.6 follows the trends of Figure 5.5, as explained before, more churn leads to more failed transactions, i.e., more updates are lost.

5.2.3 Related Work

The most relevant work on the performance evaluation or dependability analysis of data layer operations of SON under churn is found in [30]. This work evaluates the frequency of inconsistent lookups, overlapping responsibilities and unavailability of keys in Chord [10],[11] resulting from unreliable failure detectors and churn. There are two other theoretical works on churn in Chord [32, 31]. In [32] a master-equation-based approach is used to predict the performance and consistency of lookups under churn. A fluid model of Chord under churn is proposed in [31]. Our optimistic low-cost lazy data migration shows improved data-layer operations under churn using only correction-on-* as the sole maintenance, and thus can be used to complement existing works.

Part III

Stress Investigations

Chapter 6

Churn

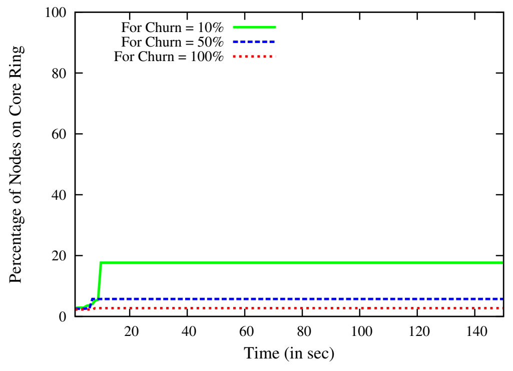
In this chapter, we focus on one property of the network, namely *Churn*, which is the rate of node turnover, i.e., nodes failing and being replaced by new correct nodes. For a peer-to-peer network, node turnover is the most usual scenario. Though, in most existing applications churn remains under a certain limit, as per studies [71, 72, 73] systems with low/average churn face high peaks. Consider the scenario of a *Structured Overlay Network (SON)* running on mobile phones or on an ad hoc network. Such a dynamically changing underlying network has still not been used because the environment is considered to be too inhospitable. The goal of this work is to explore how to build systems that are able to survive and give useful functionality for such environments. This knowledge can improve system design with enhanced self-managing properties, while opening new vistas for applications - it is also one of the future scenarios that we investigate.

We assume that churn varies over time, and that the average number of correct nodes at any instant is constant. We construct a SON that is able to survive extremely high levels of churn, and when churn returns to a low value, the functionality originally available at the low value will again be available. In other terms, the goal is to make the system reversible against extremely high churn so that at the system level, the overall functionality depends only on the current intensity of churn and not on the history of this value. High churn will cause certain functionalities to disappear and when churn decreases, they come back. Therefore there is an analogy between high churn and temporary failures, at the system level, even though at the individual node level there are only permanent failures.

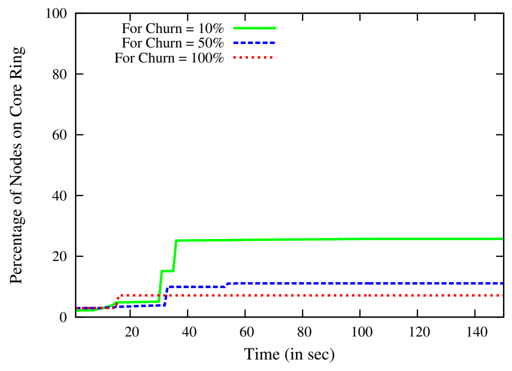
We design our SON using the self-management principles necessary to make it reversible. To our knowledge, no previous SON provides reversibility for the high values of churn we investigate. We examine the reasons why and we add the maintenance principles necessary to make it reversible. We demonstrate reversibility through simulation using realistic network conditions and churn varying over a large range [50].

6.1 Evaluation of Reversibility

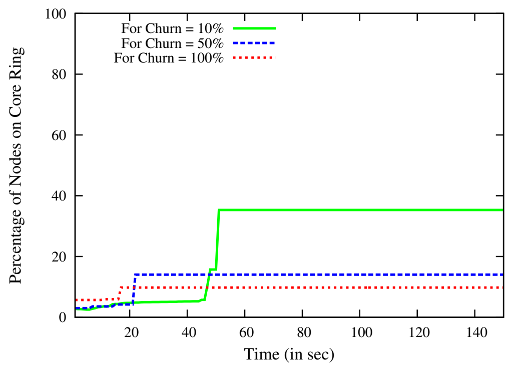
Achieving Reversibility requires Knowledge Base. Reversibility is a nontrivial property. To our knowledge, no existing work demonstrates reversibility for a SON under continuous



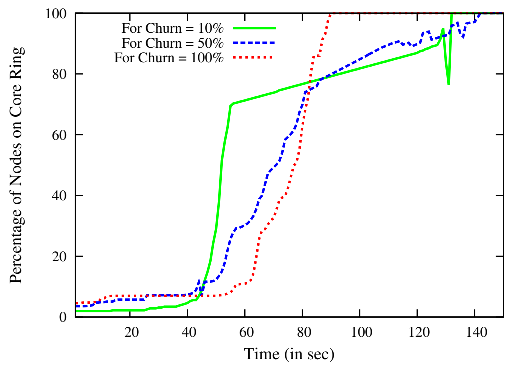
(a) Using Correction-on-*



(b) Using Correction-on-* and Periodic Stabilization



(c) Using Correction-on-*, Periodic Stabilization and Merger with Passive List



(d) Using Correction-on-*, Periodic Stabilization, Merger with Knowledge Base

Figure 6.1 – Percentage of nodes on the core ring as a function of time (in second) after withdrawing churn to assess reversibility. Figure 6.1a, 6.1b and 6.1c are not reversible (nodes on the core ring never converges to 100%). Figure 6.1d using Knowledge Base is reversible.

high churn. Our experimental results, presented in Figure 6.1, verify that the knowledge base is essential to ensure reversibility under continuous high churn. We assess reversibility in stepwise fashion, by integrating a new maintenance principle at each step and evaluating the behavior of the resulting system. We achieve reversibility only in the final step, shown in Figure 6.1d, which adds the knowledge base. We now explain these experiments in detail.

In our evaluations, we use our representative system, Beernet [13]. This is a straightforward SON that supports all the maintenance principles discussed in Section 3.3. It is an example of the reference architecture. For assessment of reversibility we use a SON of 1024 peers. All experiments are done in Mozart-Oz 2.0 [95] in a simulated environment. To simulate the underlying network, the end-to-end delays are set based on the empirical distribution of minimum RTT provided in [87]. The distribution represents significant geographic diversity, so we can say that the simulated SON is geo-distributed. We have defined churn as described in Section 4.1: percentage of nodes turnover per time unit (second in this work). If we assume equal probability of join/leave event and a single event per time unit, then every other time unit, a node will leave and a new node will join the network, i.e., every other time unit the total number of peers will be the same, whereas only a single node has a changed identity. In the steady state of SON when all nodes are on core ring, we have injected 10%, 50% and 100% churn for 1 minute. We have used 3 values of churn to compare and study healing for low, medium and high churn. The churn events are modeled as a *Homogeneous Poisson Process (HPP)* with λ events per second, where $\lambda = \frac{2*C*1024}{100}$ for $C\%$ churn on a network of 1024 peers. After withdrawing churn, we observe healing capability of SON with time. In order to quantify the effect of self-healing, we have used metric: percentage $(\%)$ of nodes on core ring, to measure the rigidity of the ring. In other terms, we find out the maximal ring in the system and report percentage of nodes on it. The ultimate goal is to have the metric converge to 100%. A fixed workload is used in an experiment by injecting transactions, modeled as an HPP with $\lambda = 1$ transaction per second. A transaction reads one key and updates another one. Since the withdrawal of churn, for each second we present percentage of nodes on core ring and an average of 20 independent runs are taken for each second. We remark that the termination time of an experiment in Figure 6.1d, apparent in the result, is the maximum among the samples used.

6.1.1 Correction-On-Change and Correction-On-Use

As Figure 6.1a shows correction-on-* principles fail to achieve reversibility even for the lowest intensity (10%) of churn used in our experiments. Correction mechanisms allow the pointers to get fixed as soon as a failure, leave or join is detected, rather than waiting for a periodic check. Such rapid response reduces the probability of inconsistency and has more efficient bandwidth consumption. However, after withdrawing churn, the structure of the system remains almost the same. This is due to the lack of liveness of these principles. As healing is only triggered whenever a join/fail event occurs, so after churn is withdrawn there is no such event to continue the healing process. Also, under churn, a node based on its current state handles a new event, whereas the current state of the node might already be not optimized, for example, it might be on a branch and have missed an opportunity to reduce branch size. In such situation, handling high churn aggravates the structure, by unnecessarily pushing more and more nodes on branches. Although Beernet allows branches, but increasing branch size affects routing efficiency and increases the probability of creating isolated branches, thus inconsistencies under churn, as explained in [13]. So it is ideal and required to make branches shorter by bringing nodes on the ring and make the ring perfect whenever possible. But the branch-pruning algorithm is triggered only when a node joins, making it less effective. A node only heals whenever it senses an event based on its current state and later if a correction is required there is no option to trigger or propagate that, thus the damage of the structure remains as it is. So the healing capability, thus reversibility of these principles is limited in the face of extremely high churn.

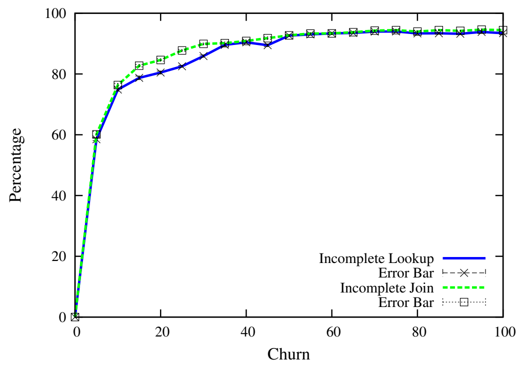


Figure 6.2 – Percentage of lookups and joins which remain incomplete after injection of churn for 1 minute

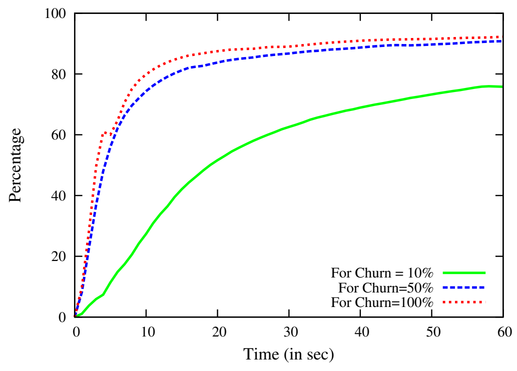


Figure 6.3 – Percentage of incomplete joins with time during injection of churn for 1 minute

6.1.2 Correction-on-* and Periodic Stabilization

We can see improvements in Figure 6.1b, after integration of periodic stabilization; however still unable to achieve reversibility. The period used in our experiments for periodic stabilization is 3 seconds. Each node independently performs periodic maintenance by exchanging messages with its successor to maintain the local geometry. Due to such periodic maintenance, nodes on branches eventually become part of the core ring. The periodic stabilization thus provides a shortsighted vision for each node about the current state of its immediate neighborhood. However the healing is taken place only during first few rounds, and after that there is no change in the SON structure, as shown in Figure 6.1b.

6.1.3 Correction-on-* and ReCircle (Periodic Stabilization and Merger with Passive List)

As Figure 6.1c shows, the integration of merger with passive list does not show much improvement over the combined local healing. The reason is the existence of isolated nodes in the system (explained below). The nodes on the overlay have no reference to these nodes, i.e., no suspicion is issued. As a result, reactive merge attempts gain no success. The period used to dequeue the elements to generate *mlookups* at each node is 5 seconds and we have kept the *fanout* parameter as 1. Though the integration of this reactive global maintenance is unable to achieve reversibility, the knowledge is navigated further in the identifier space, triggering local maintenance during subsequent rounds of periodic stabilization.

Why not reversible yet? In Figure 6.1c all nodes are still not on the core ring, i.e., the system is still not reversible. As our investigation shows, there are peers whose joining processes fail under churn. With the increment of churn more peers are unable to join the network. This is most likely a phase transition at system level, as discussed in Chapter 10, where continuous high churn injection makes the relaxed ring unstable that do not allow new peers. The first step of joining a SON is to do a *lookup* for successor and after receiving a response the new peer becomes part of the SON. For the join to fail, there are two possible reasons: i) the lookup request is lost while routing, ii) the successor has failed after receiving the lookup request; in both cases the new peer keeps on waiting for response. If we ignore the processing time at successor, then the probability of join failure is proportional to lookup failure. Figure 6.2 shows percentage of incomplete lookups and joins for different values of churn. We have used the same experimental setup of 1024 nodes and injected increasing churn for 1 minute. Simultaneously, lookup requests are created using a HPP with $\lambda = 100$ requests per second. After waiting for another 30 second, we report percentage of lookups which remains unanswered, also Figure 6.2 shows percentage of join requests which are not yet complete. As churn increases more and more join requests remain pending. We also present the accumulation of pending join requests with time in Figure 6.3, especially for high churn. As it is evident, with high churn, the join request of the new peer is lost.

6.1.4 Correction-on-*, Periodic Stabilization and Merger with Knowledge Base

As shown in Figure 6.1d, after integration of the knowledge base the system has achieved reversibility. As the churn intensity increases, more and more join requests remain pending, which isolates these peers. In order to ensure successful joining of new nodes, we have extended nodes with repeated join attempts (until a response is received) with new join references. This join reference, i.e., knowledge about an alive peer on the overlay is gathered from the distributed knowledge base. In our experiments, if a node is unable to join with its current join reference within 90 seconds, then it requests a new join reference from the application layer. The application layer provides a new join reference by accessing and accumulating the distributed Knowledge Base, or using a previously cached one. The isolated peer then triggers a new join request with that. The waiting period for an isolated peer, to receive a response of its join request with a join reference, is a tunable parameter, which we will refer as *Join Timeout*. We have chosen a conservative value of 90 seconds for this parameter to avoid triggering of unnecessary repeated join requests. This parameter can be adapted based on the operating environment and RTT distribution of the underlying network, which we leave as future work. Along with this, we have used the proactive manner of triggering the merger using Knowledge Base. In some runs we have observed partition of the system after the isolated nodes complete their join procedures. For these scenarios, the PL approach used in [37] (as discussed in Section 3.3) fails to trigger the merging. In order to merge such partitions proactive global maintenance using Knowledge Base is required. As we can see in Figure 6.1d, for all of our 20 runs for each churn intensity, the damage of the structure caused by churn is completely healed; thus the system is reversible.

6.2 Evaluation of High-Level Properties

We evaluate how functionality (e.g., transactions, storage, replication) decreases as churn increases. Again, we use the Beernet [13] system, which provides strong consistency with transactions, key-value storage, and replication. We have analyzed the following metrics:

- Damage to the ring topology in terms of percentage of nodes on core ring;
- Topology recovery/healing time and cost;
- Data level parameters: percentage of failed transactions and percentage of lost keys, percentage of inconsistent replicas, percentage of lost updates.

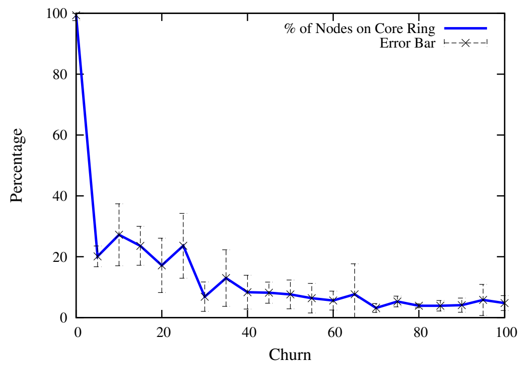
We use a similar experimental setup as described in Section 6.1. We bootstrap a SON of 100 peers. Correction-on-*, periodic stabilization and merger with Knowledge Base are used as part of maintenance. The underlying network is simulated based on the empirical distribution of minimum RTT provided in [87]. We have used a stream of transactions to create workload, where transactions are modeled as a HPP with $\lambda = 1$ transaction per second. A transaction reads one key and updates another one. During steady state of the SON, churn of increasing intensity is injected. For each value of churn, the system experiences join/leave events for 1 minute. The churn events are also modeled as a HPP, as explained before. We have used mean value of 10 independent runs for every 5% increase of churn.

6.2.1 Damage and Recovery of Ring Topology

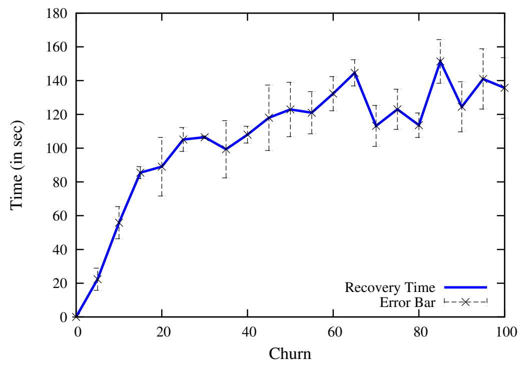
After churn is withdrawn; we present the snapshot of the ring topology (in terms of percentage of nodes on core ring) of the overlay in Figure 6.4a. This gives an idea the damage done to the topology due to increasing churn. We also measure the time (in seconds) required for the existing nodes to organize into a perfect ring topology, once churn is withdrawn. Figure 6.4b presents the recovery time or time to heal the damage (as shown in Figure 6.4a) of ring topology for increasing churn. As expected, it requires more time for the SON to regain/restore its topology with increasing churn. We also present the number of messages generated for each value of churn in Figure 6.4c, which provides an approximation of bandwidth consumption for the self-healing of our system. We find in Figure 6.4c that the number of messages is higher for lower churn. The reason behind this is that for lower churn there are less isolated nodes, i.e., almost all the nodes are part of the SON. These nodes periodically issue maintenance messages, whereas an isolated node does not have a reference to any other node, so no periodic message is exchanged. This, in fact, reflects the impact of costly periodic stabilization. In order to make the overall maintenance efficient and scalable, it is essential to adapt with the operating environment, which we keep as future work.

6.2.2 Data Level Parameters

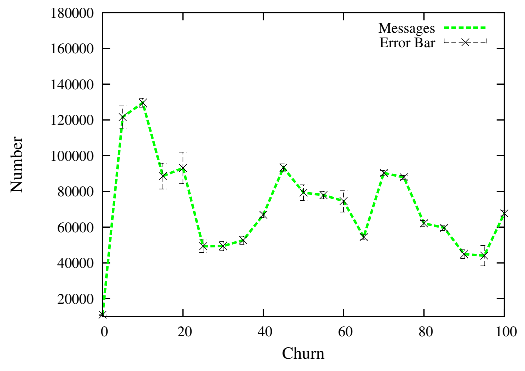
To understand how the data-layer performs during a challenging environment like extremely high churn we present percentage of failed transactions in Figure 6.5. As expected, as the intensity of churn increases, almost all transactions abort. After churn is withdrawn we do a read transaction to read all data items in order to understand the impact of the



(a) Damage of the topology



(b) Recovery/Healing time



(c) Number of messages during churn and recovery

Figure 6.4 – Properties for increasing churn after injecting a particular churn value for 1 minute

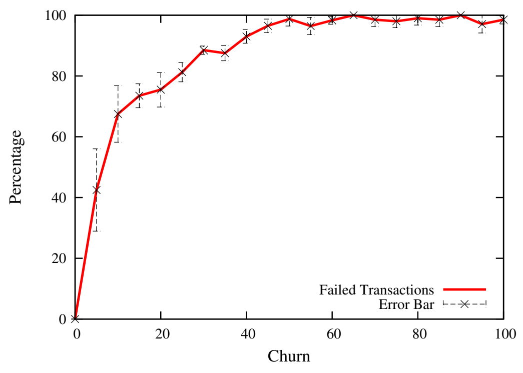


Figure 6.5 – Percentage of failed transactions

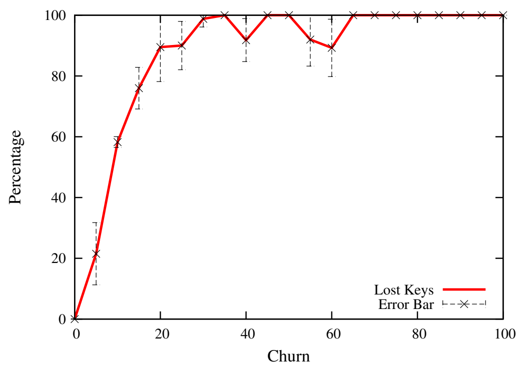


Figure 6.6 – Percentage of lost keys

harsh environment on the data storage and report percentage of lost keys in Figure 6.6. After churn is beyond 30%, almost all keys are lost. From this result, we can conclude that in order to survive in stressful operating environment, the application design needs to take into these issues into consideration. The remaining two parameters, namely fraction of lost updates and inconsistent replicas, follow the trends of percentage of failed transactions and lost keys respectively.

6.3 Related Work

The work in [29] mainly focuses on routing level correctness or consistency and improved performance of a SON. The evaluation of MSPastry (a new implementation of Pastry[96]) is conducted by varying environmental parameters like network topology, node session times, link loss rates, and amount of application traffic. This work validates proposed techniques for improved routing performance and dependability, in the face of high churn. Another work on lookup consistency, [30] evaluates the frequency of inconsistent lookups, overlapping responsibilities and unavailability of keys in Chord [10],[11] resulting from unreliable failure detectors and churn. Krishnamurthy et al. in [31] use fluid model approach to do a theoretical analysis of Chord [11] under churn. They present the functional form of the probability of network disconnection and the fraction of incorrect pointers (successor and fingers). In [32] a master-equation-based approach is used to predict the performance and consistency of lookups under churn. Also, in their continuing analytical study in [33] use the analytical tool based on master-equation approach of physics to do a comparative analysis of periodic stabilization and correction-on-change maintenance principles under churn. Another analytical work in [34] establishes a lower bound on the maintenance rate of a SON under churn in order to remain connected. In [45] and [46], El-Ansary et al. use a physics-inspired analytical approach to analyze performance of large scale distributed systems and also investigate intensive variables (i.e., variables which are independent of system size) related to self-organization and self-repair. They apply this methodology for Chord and propose an intensive variable to describe the characteristic behavior of the overlay. Apel et al. [97] describes design decisions such as self-tuning mechanisms based on software-engineering principles for self-organization/self-adaptation of overlay networks. The analytical framework presented in [35] can be used to characterize the routing performance of SON under churn. Our investigation of churn is an empirical study that can complement these analytical works.

6.4 Discussion

In this chapter we investigated the impact of extremely high churn at various level of the system. We conclude that in order to design large-scale distributed systems for highly dynamic operating environments, like edge network, the inherent high node dynamicity must be taken into account. To enable the system survive, yet able to provide useful functionality, in such operating environments, reversibility of the system must be ensured. This chapter presents the construction of a healing mechanism, which can make the system reversible in case of extremely high churn or burst of high churn. For this purpose we have integrated different principles and assessed the enhancement of healing capability after each extension. Also, we have analyzed the shortcomings of each principle. To summarize the outcome of our experiments: *Efficient* maintenance strategies fail to achieve reversibility as churn increases; a *Resilient* maintenance strategy is required to make the system reversible in case of extremely high churn. Also, we remark that as churn causes perturbation of the global state of the system, the strongest form of a resilient maintenance strategy, i.e., unrestricted gossiping is essential to achieve weak reversibility of the system after a burst of high churn.

Chapter 7

Network Partitioning

In this chapter, we investigate a particular operating environment of distributed systems, namely *Network Partition/Merge*. During the partition of the underlying network, the nodes of a system are divided into multiple disjoint sets, where a node can communicate with the nodes of its own set, but is unable to contact the nodes in the other sets. Any long-running large-scale distributed system is bound to come across network partitioning during its execution. Consider the scenario of a SON running on mobile phones or on an ad hoc network, which has high node mobility and intermittent connectivity, and undergoes frequent changes in network topology. In such a dynamically changing environment network partitioning can be a frequent event. Due to self-* properties (mostly partial), most of the ring-based SONs are expected to provide partition tolerance by forming separate overlay(s) in each partition. However, once the partition ceases (we will term this event as *Network Merge*), the system should reverse back, i.e., merger of multiple overlays, resulted from endurance during partition. In this chapter we evaluate partition tolerance and merging (as the network partition ceases) capability or reversibility of the system using existing maintenance strategies, namely Correction-on-Change, Correction-on-Use, Periodic Stabilization, and Ring Merge. We identify the necessary and sufficient maintenance strategies to ensure partition tolerance for any scenario of network partitions [98]. By means of simulations, we demonstrate reversibility, once network partition ceases, for overlay networks with high levels of partition and we make general conclusions about the ability of the maintenance strategies to achieve reversibility for the system [98]. In this chapter, we only consider scenarios where no churn is induced during partition, which corresponds to network partition of short duration.

7.1 Types of Partition

There can be two different types of partitioning of the overlay that may arise as a result of the underlying physical network partition. This differentiation happens due to the *locality-awareness* of the mapping function F_p, which associates peers with a unique virtual identifier from the identifier space (Section 3.1). If F_p is locality-aware, then logically adjacent nodes (on the overlay) are also physically close; e.g., for DKS F_p is order-preserving to

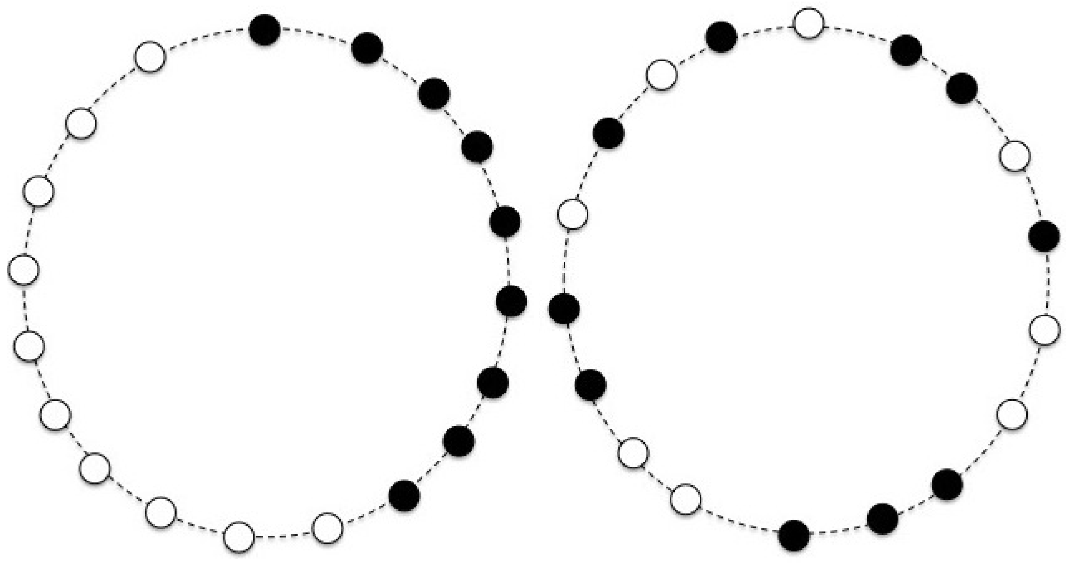


Figure 7.1 – Two different types of partition scenarios: white and black nodes belong to two different partitions

ensure that nodes in same organizational domain are logically close on the overlay. As a result, an underlying network partition usually divides such overlay into P contiguous regions (here, P is the number of partitions). We will refer this particular kind of partitioning as *Sequential Partitions*, which emphasizes locality. Figure 7.1a shows an example of sequential partitions. On the other hand, if F_p is not locality-aware or some random function then logically adjacent nodes can be very far apart physically, e.g., for Chord, F_p is a uniform hash function, which uniformly distributes nodes on the identifier space. For such mapping, an underlying network partition might cause adjacent nodes on the overlay to be in different partitions. We will refer such partitioning as *Sparse Partitions*, which emphasizes non-locality. In Figure 7.1b, we can see sparse partitions, where adjacent nodes tend to be in different partitions.

7.2 Evaluation of Maintenance Strategies

We distinguish two different cases for a SON, corresponding to a network partition: 1) execution during network partition (i.e., partition tolerance) and 2) partition repair (i.e., achieving reversibility). Existing works do not analyze case 1 in depth, have only mentioned the particular partition scenarios for which a maintenance strategy is able to provide partition tolerance [13]. Shafaat et. al. [37] do not address case 1, their work is about merging of multiple ring overlays, i.e., case 2. We have identified the preconditions to ensure partition tolerance for any possible partition scenario. For case 2, there are several works on the ring merge algorithm [37], [38], but no work has been found that assesses partition

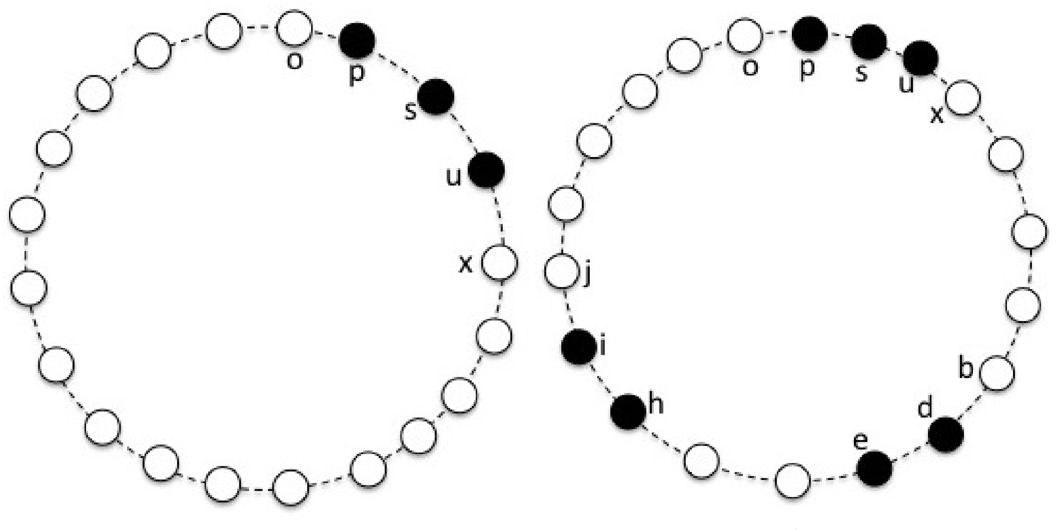
Scenario	Local Maintenance (Correction-on-*, Periodic Stabilization)	Global Maintenance (Merger with Passive List/Knowledge Base)
Execution During Network Partition	Can Create separate rings in each partition but can get stuck: multiple rings can be formed in the same partition	Merger with KB is Required to provide the best partition tolerance; however Merger with passive list can fail to fulfill the requirement
Execution At Partition Repair	Combined reactive and proactive corrections is able to merge multiple overlays, even reacts quicker	Provides no improvement over the combined local strategies

Table 7.1 – Self-healing achieved by Maintenance Strategies

repair capability of local maintenance principles and that compares their performance with an explicit merger. We have analyzed ring merge capabilities of correction-on-* (with and without periodic stabilization) and have shown that integration of an explicit merger gains no significant improvement if no churn is induced during partition. Table 7.1 summarizes our result. In order to have the best partition tolerance during a network partition, global maintenance is required along with local corrections. To repair partitions, combined reactive and proactive local corrections are enough to achieve complete self-healing, which even shows quicker response as the network partition ceases.

7.2.1 Execution During Network Partition

In order to ensure partition tolerance for any partition scenario, proactive global maintenance, i.e., the merger with KB is required along with local ones. During partition it is impossible to achieve both global consistency and availability as per the *CAP* theorem [99], however each ring should be consistent in itself. In our work we refer a SON to be partition tolerant (or survive a partition) iff i) the peers on each partition will form a separate overlay, ii) the system shows high availability i.e. every lookup must result in a response and iii) each overlay is consistent in itself. Both correction-on-* and periodic stabilization is able to provide partition tolerance as long as every peer is able to find a valid successor candidate in its successor list, as also identified in existing literature. We will analyze the partition tolerance capability of these two local maintenance strategies, when this condition does not hold, using the example scenarios in Figure 7.2.



(a) (b)

Figure 7.2 – Two partition scenarios: white and black nodes belong to two different partitions; partition having black nodes have absence of more than $|succ\_list| - 1$ consecutive peers (here, $|succ\_list| = 4$).

Local Maintenance

As already mentioned, both correction-on-* and periodic stabilization are able to survive a network partition as long as no more than $|succ\_list| - 1$ consecutive peers are absent from a partition. Comparing two different partition types, described before, sequential partitions are more prone to face scenarios, where this condition is not satisfied, than the sparse partitions.

Using only correction-on-* is insufficient to provide partition tolerance for the partition scenarios in Figure 7.2. When peer u suspects its current successor x due to network partition, it fails to find a valid successor in its successor list to trigger the failure recovery, as the next consecutive 5 peers are partitioned away (which is $> |succ\_list| - 1 = 3$), thus sets its successor pointer as itself. In a similar way, peer i also fails to find its successor in Figure 7.2b. Peer p (and peer d in Figure 7.2b) suspects its current predecessor due to partition, but as nobody triggers the recovery mechanism, the responsibility of peer p (and peer d in Figure 7.2b) doesn't change, resulting in gap in the identifier space. The situation remains the same until the partition ceases. This introduces unavailability of keys in the range $(u, o]$ in Figure 7.2a and $(u, b]$, $(i, o]$ in Figure 7.2b for the partition holding black nodes. Also, the nodes fail to form an independent overlay in this partition as per the predefined structure or embedded graph. So, correction-on-* fails to survive these partition scenarios.

Using periodic stabilization improves availability, but multiple ring overlays are formed in the same partition. For Figure 7.2a, peer u is unable to find a valid successor in its successor list after the partition, and therefore sets its successor pointer as itself. During the next round of periodic stabilization (suppose $round = t$), s asks u about its predecessor. On receiving this message, u sets s as its successor, as s is a better successor than itself. During $round = t + 1$, u asks s about its predecessor and comes to know about p. As p is a better successor for u, u sets its successor pointer as p, which also triggers change of responsibility of p. Thus, eventually p, s and u form a separate overlay in this partition and the gap in the range $(u, o]$ is healed. Though temporary unavailability of a certain range of keys do occur, it is possible to eventually overcome that, where peers on each partition form an independent overlay, which is consistent in itself and provides availability. However, for the partition scenario in Figure 7.2b, this conclusion does not hold entirely. As we can see in Figure 7.2b, there are two instances of minimum $|succ\_list| - 1$ consecutive peers absent from the partition holding black nodes. As in Figure 7.2a, peers p, s and u form a ring, so do peers d, e, h and i. So, there are two ring overlays formed in the same partition and remains the same, until the partition ceases. The reason is that periodic stabilization only does local healing around a node's immediate vicinity and lacks the mechanism to spread out the healing globally. So, for the partition scenario in Figure 7.2b, the healing of periodic stabilization falls short to satisfy the first condition of partition tolerance (as described before), where there is more than one overlay formed in a partition (and remains the same), though the nodes on the overlays are able to communicate with each other. As per our analysis, during a network partition, the number of overlays formed in the same partition is equal to the number of instances of minimum $|succ\_list| - 1$ consecutive missing peers from that partition.

Global Maintenance

As described before, local maintenance can get stuck while providing partition tolerance. For example, for the representative partition scenario in Figure 7.2b, periodic stabilization eventually organizes the nodes in the partition (holding the black nodes) into two different overlays. In order to merge these two overlays, we need an overlay merge algorithm, e.g. the merger of ReCircle. If any peer from one overlay is enqueued into the queue of a peer of another overlay, the merger will be triggered, thus eventually resulting into a single overlay in the partition (holding the black nodes). Using the passive list approach for this purpose fails to trigger the merging process, as nodes in the same partition are not suspected by each other. We have experimentally verified this result for the representative partition scenario in Figure 7.2b. However, triggering merger in a proactive manner using a knowledge base is able to provide the desired outcome, as also validated via simulation. By exploiting the knowledge base at each node periodically (as described in Section 3.3.5), eventually the merger will be triggered as a result of enqueuing a valid peer of other overlay, with which communication can be established, thus resulting into a single overlay in the partition (holding the black nodes).

7.2.2 Execution at Partition Repair (Network Merge)

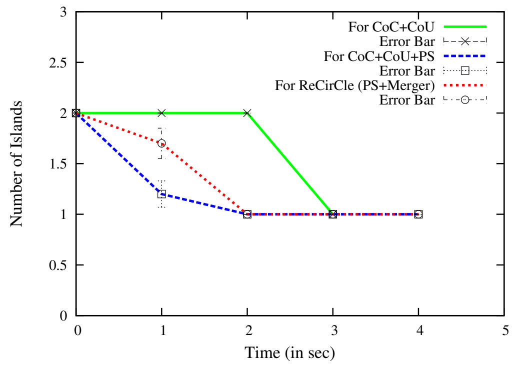
As a partition of the underlying physical network ceases, which we refer as network merge, the multiple overlays formed during partition, should also be merged i.e. reverse back to original state. In this section, we assess reversibility of each maintenance strategy and also find the limit (if any), while a network merge happens.

We use a SON of 100 peers. All experiments are done in Mozart-Oz 2.0 [95] in a simulated environment. To simulate the underlying network, the end-to-end delays are set based on the empirical distribution of minimum RTT provided in [87]. The distribution represents significant geographic diversity, so we can say that the simulated SON is geo-distributed. We simulate partitioning of the underlying network in order to create inhospitable operating condition. In the steady state of SON when all nodes are on core ring, we simulate 2, 4 and 10 partitions of the SON, of almost equal sizes. The partition of the SON is withdrawn after 30 seconds and we observe the merging of overlays with time. We have used three scenarios, with increasing number of partitions, in order to compare and study the reversibility based on number of partitions.

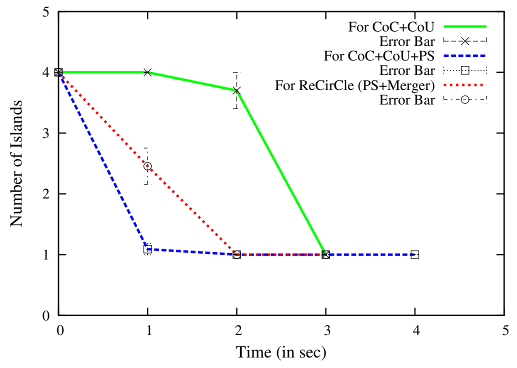
In order to quantify the effect of self-healing, we have used metric: Number of *Islands*. We define an island to be a disconnected sub-graph by following the successor pointer of each node. The number of islands can be $\geq$ number of physical partitions, due to temporary inaccuracy of the successor pointers or inadequate healing of the maintenance strategy (e.g., as explained in Section 7.2.1, for Figure 7.2b, 3 islands are created, though the number of partitions is 2). The value of this metric should be 1, for a single run, if overlays are merged together after the partition disappears. However, we still continue an experiment until all nodes form a perfect ring, i.e., complete self-healing. Although our representative system, Beernet, allows branches, but increasing branch size affects routing efficiency and increases the probability of creating isolated branches, thus inconsistencies under churn, as explained in [13]. So, for a single run, the ultimate goal is to make “number of islands” as 1 and all nodes to be on the core ring. We present the number of islands against time (in second) after the partition disappears and an average of 10 independent runs are taken for each second. It is noticeable that, the termination time of an experiment, apparent in the result, is the maximum among the samples used. A fixed workload is used in an experiment by injecting transactions, modeled as a Homogeneous Poisson Process with $\lambda = 1$ transaction per second. A transaction reads one key and updates another one.

Sparse Partitioning

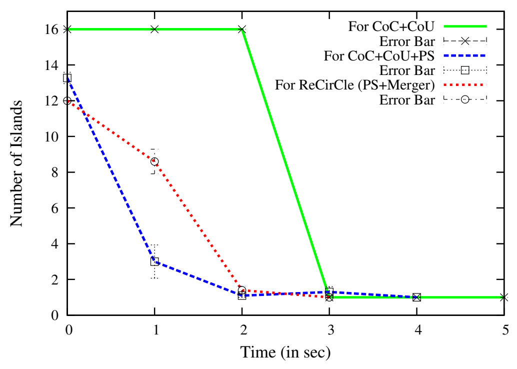
In this section we present reversibility results for sparse partitions. Sparse partitions create higher stress on the merge operation than the sequential partitioning. In our simulation, to create P sparse partitions, we create P baskets, where each node of the SON is assigned to a basket with probability $\frac{1}{P}$. The comparative analysis of healing capability of maintenance strategies is portrayed in Figure 7.3 as the average number of islands with time, since the network merge happens. Also, the average number of messages generated for each experiment is shown in Figure 7.4. As we can see in Figure 7.3, the combination of correction-on-* and periodic stabilization is sufficient to achieve reversibility if there is no churn. Merger with passive list, as done in [37], does not show improvement over the combined local maintenance strategies.



(a) Number of Partitions=2



(b) Number of Partitions=4



(c) Number of Partitions=10

Figure 7.3 – Number of islands as a function of time (in second) starting at the moment of sparse partition repair to assess self-healing using different maintenance strategies

Local Maintenance: Correction-on-*, due to its rapid reaction against an event (join/leave/-failure/false suspicion) exhibits certain overlay merging capabilities, however, fails to complete the healing process when the number of partitions goes higher. We can see in Figure 7.3, irrespective of the number of partitions created, correction-on-* has created 1 island once the partition ceases. The reason is that, as a result of partition, each node considers the nodes on other partitions to have failed and adjusts its pointers, however, it continues monitoring those suspected peers. As no churn is experienced, so the list of monitored nodes is not altered. So when it can make a connection with those suspected peers (as partition disappears), a false suspicion event (see Section 3.3) is triggered, as a result of which it corrects its pointers, resulting in merging of overlays. However, as we can see in Figure 7.3c, though the number of underlying partition simulated is 10, there are 16 partitions of the overlay (at $t = 0$), justifying the explanation presented in Section 7.2.1. In terms of overlay merging capability, the correction-on-* has made the number of islands as 1, however it fails to make a perfect ring for most of the runs. An average of 91.5% nodes are on core ring for those runs, where rest of the nodes are on branches. While merging large number of overlays, nodes are placed on branches as an initiation, but as the events (detection of false suspicions in this case) fade away, healing is also discontinued, resulting in branches to remain in the system. However, this is the least-cost healing mechanism as a response of network merge, as shown in Figure 7.4. In comparison with other maintenance principles, the load created on underlying network is much lower for these principles, making it a cheap healing response against any inhospitable event, without creating any inconsistency.

Periodic Stabilization itself is not capable of merging multiple overlays, as also supported by [11]. So, we have combined periodic stabilization with correction-on-*. The period used in our experiments is 1 second. As we can see in Figure 7.3, the combination of reactive and proactive mechanisms show quick response among all. In terms of partition tolerance, we can see the improvement in Figure 7.3c comparing to the correction-on-* mechanisms, however the number of partitions is still more than the induced one, reason for this is explained in Section 7.2.1. The impact of the integration of costly periodic stabilization is clearly visible in Figure 7.4. However, the number of messages decreases with the increase of number of partitions. The reason is that, with the increase of the number of partitions, the size of each independent overlay decreases, resulting in less number of messages to propagate any correction. Thus, any correction within fewer rounds covers an entire small overlay (with fewer nodes) while enduring network partition.

Global Maintenance: As shown in Figure 7.3, the integration of a global maintenance, e.g., merger with passive list, does not show any significant improvement over the combined local healing. The period used to dequeue the elements to generate *mlookups* at each node is 3 seconds and we have kept the *fanout* parameter as 1. However, as we can see in Figure 7.4, this is one of the costliest solutions, even with the *fanout* as 1, especially while merging larger number of overlays. Although the number of messages is lower for 4 partitions than 2, the reason is due to propagation of corrections by periodic protocol across smaller overlays, as described before, but this effect is overcome by the reactive messages generated in case of 10 partitions. Also, the number of messages generated in case of 2 and

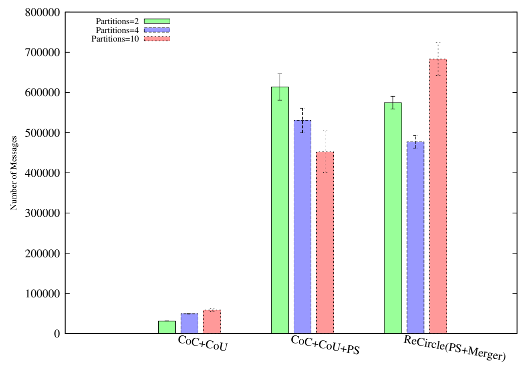


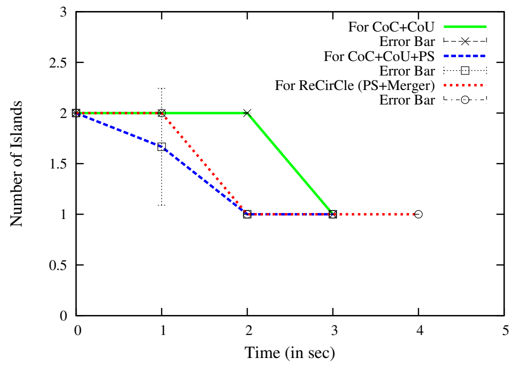
Figure 7.4 – Number of Messages generated for 2, 4 and 10 sparse partitions using different maintenance mechanisms

4 partitions are lower than those for the combined local strategies. The reason is the global spreading of corrections by the merger.

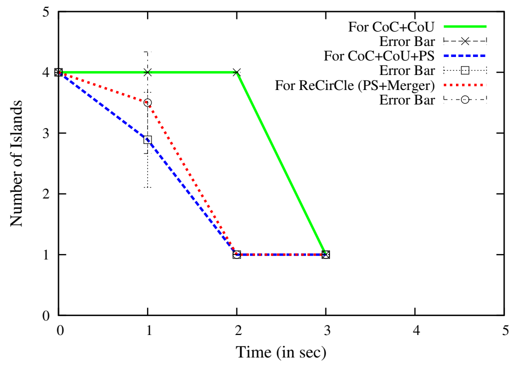
Sequential Partitioning

In this section we investigate overlay merging as a network partition ceases, when the partitioning is sequential. Sequential partitioning causes the overlay to split into P contiguous regions (here P is the number of partitions). As already mentioned, merging of a sequential partitioning creates much less stress on overlay maintenance than a sparse partitioning. In fact, as a network partition disappears, only 2 nodes in each partition need to modify their pointers, if no churn is experienced during the partition. In our simulation, to create P sequential partitions, we create P baskets, where N/P contiguous nodes on the overlay (by following the successor pointer of each node) are assigned to each basket, here N = total number of nodes on the overlay = 100 in our experiments. Figure 7.5 shows the comparative analysis of reversibility of maintenance strategies. We have used the same metric to assess self-healing: average number of islands with time, since the network merge happens. Also, Figure 7.6 presents the average number of messages generated for each experiment. As we can see in Figure 7.5, all our experiments have achieved complete self-healing irrespective of the number of partitions of the overlay. However, as with sparse partitioning, the combined local maintenance (correction-on-* and periodic stabilization) shows quick response among all. The integration of a reactive global maintenance, e.g., merger with passive list , as done in [37], does not show any improvement over the combined local maintenance strategies.

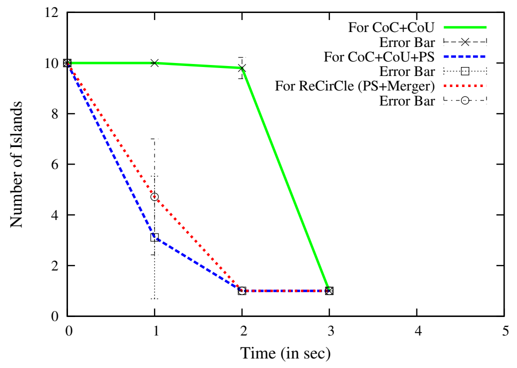
Local Maintenance: Unlike sparse partitioning, while repairing sequential partitions, the reactive local maintenance, correction-on-* principles are able to achieve reversibility even for higher level of partitioning, as evident in Figure 7.5. The reason is the same as described for sparse partitioning, the rapid reaction of these principles against any event (join/leave/-failure/false suspicion). As we can notice in Figure 7.5, the number of partitions in the



(a) Number of Partitions=2



(b) Number of Partitions=4



(c) Number of Partitions=10

Figure 7.5 – Number of islands as a function of time (in second) starting at the moment of sequential partition repair to assess self-healing using different maintenance strategies

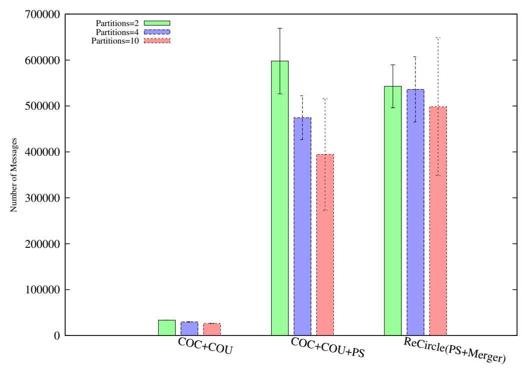


Figure 7.6 – Number of Messages generated for 2, 4 and 10 sequential partitions using different maintenance mechanisms

system at $t = 0$, is the same as the number of simulated partitions of the underlying network. However, correction-on-* principles fail to provide partition tolerance for any of these scenarios, as unavailability of key ranges are introduced, also the nodes in each partition fail to form a ring overlay in each partition, as described in Section 7.2.1. In terms of overlay merging capability, the correction-on-* principles are sufficient for sequential partitioning of any number, even offer the least-cost complete self-healing among all, as portrayed in Figure 7.6. Though these principles show slowest response among all, but the results obtained is consistent irrespective of the number of partitions in the system, i.e., for all the runs, the healing is completed within 3 time unit, since the network partition ceases.

Figure 7.5 shows, as with sparse partitioning, the combination of correction-on-* and periodic stabilization continues the trend of showing the quickest response among all while merging sequential partitions. The period used in our experiments is 1 second. This combined local maintenance provides partition tolerance for all the runs of all levels of partitioning. In terms of reversibility, this combination provides the quickest response and convergence among all. However, this is one of the costliest self-healing, as evident in Figure 7.6, making it explicit the bandwidth consumption of the costly periodic stabilization.

Global Maintenance: As Figure 7.5 shows, the integration of a global maintenance, e.g., merger with passive list, with periodic stabilization does not show any improvement over the combined local healing. On the contrary, at times the convergence time (see Figure 7.5a) and bandwidth consumption (see Figure 7.6 for 4 and 10 partitions) are more than the combined local healing. The period used to dequeue the elements to generate *mlookups* at each node is 3 seconds and we have kept the *fanout* parameter as 1.

7.3 Related Work

Several versions (through gradual improvement) of overlay merge algorithm are proposed in [37], of which ReCircle (described in Section 3.3.4) is adapted in Beernet for the purpose of this work. The evaluation, carried out in these works, is to validate the proposed algorithms and sensitivity analysis of different parameters of these algorithms on convergence time and bandwidth consumption. The objective of our work is to assess self-healing achievable using different maintenance strategies and also to identify the properties/limit of the maintenance operation to survive in an inhospitable environment caused by high level of network partition. So, the work in [37] and our work complement each other.

In [38], a centralized approach using bootstrap server is proposed to detect multiple overlays and initiate merge, as the underlying network partition ceases. As per this approach, a peer with the smallest virtual identifier, periodically sends message to a bootstrap server. As the bootstrap server receives multiple messages, it detects multiple overlays in the system, which then informs all peers to initiate the merging process. This approach depends on a central bootstrap server. Also, this work lacks a full algorithm and evaluation of merge process. The protocols for network partitions and merge are proposed in [39], at the core of which is a broadcast protocol. A node sends broadcast message to d uniformly selected nodes to gather knowledge about the network it belongs to, each node, which receive such broadcast message repeats the process. However, this work does not specify how a node acquires knowledge about the broadcast candidates and satisfy uniform sampling. Also, the merge protocol presented states that nodes of one overlay join the other, does not mention the process to detect a network merge and how to decide which overlay to trigger the join process. Merging of multiple P-Grid [100] SONs is presented in [42], [41], which is complementary to our work. Methods/algorithms to merge two independently bootstrapped peer-to-peer overlays are presented in several works [101], [102].

7.4 Discussion

In this chapter, we have investigated network partitioning. We have considered scenarios, where no churn is experienced by the overlay during a network partition, which correspond to the network partitions of short durations. We have presented comparative analysis of partition tolerance and reversibility of existing maintenance strategies, namely, correction-on-*, periodic stabilization and ReCircle (as done in [37]). We have observed that the number of partitions of the overlay (i.e., logical partitions) can be more than the number of actual partitions in the physical network (i.e., physical partition). Thus, a partition can occur even if there is no communication problem. We have identified and verified the preconditions to ensure partition tolerance for any scenario of network partition. Our results show that when the global state of the system is not perturbed, i.e., the system does not experience any churn during the partition (though communication problems might cause individual nodes to change their local states, e.g., change of successor and/or predecessor pointers), micro-level interactions among nodes (i.e., local corrections at each node) are able to trigger and accomplish macroscopic healing (i.e., merging of multiple overlays). However, the local corrections need to be both proactive and reactive to attain better recovery, especially when there are high levels of sparse partitioning of the overlay. The result obtained only through such local corrections, without any explicit merge algorithm, is competitive (or at times better) with the one with an explicit overlay merger.

Chapter 8

Interactions between Network Partitioning and Churn

In Chapter 7, we have considered network partition and merge, where there was no churn in between, which is not a realistic scenario, since for a peer-to-peer network churn events are the most usual ones. Though in most existing applications churn remains under a certain limit, as per studies [71, 72, 73], systems with low/average churn face high peaks and this may happen even during short duration of a network partition. Consider the scenario of a *Structured Overlay Network (SON)* running on mobile phones or on an ad hoc network. In such a dynamically changing environment network partition can be a frequent event, along with high churn. However, we have not found any work in literature that demonstrates reversibility for such inhospitable environment, where a SON goes through a network partition, while facing churn at the same time.

In this chapter, we propose a model, namely "Stranger Model", to generalize the impact of *simultaneous* network partition and churn [98]. We show that this interaction causes partitions to eventually become strangers to each other, which makes full reversibility impossible when this happens. Using this model, we can predict when irreversibility arrives, which we verify via simulation. Later, we evaluate the reversibility of maintenance principles while facing churn during a network partition and identify the preconditions to achieve reversibility [98]. In this chapter, we have used only sparse partitioning for our experiments, as sparse partitions create higher stress on the maintenance than the sequential partitioning.

8.1 Stranger Model

Before the network partition and after network merge, churn is handled by any SON as usual. It is churn during a network partition, which creates a challenge for merging the overlays as network merge happens. We propose a model, namely "Stranger Model", to understand the impact of churn during a network partition. Using this model we quantify the challenge for the maintenance mechanism, while merging multiple overlays (created while enduring the network partition), as the partition ceases.

We use the same definition of churn as in Section 4.1: percentage of nodes turnover

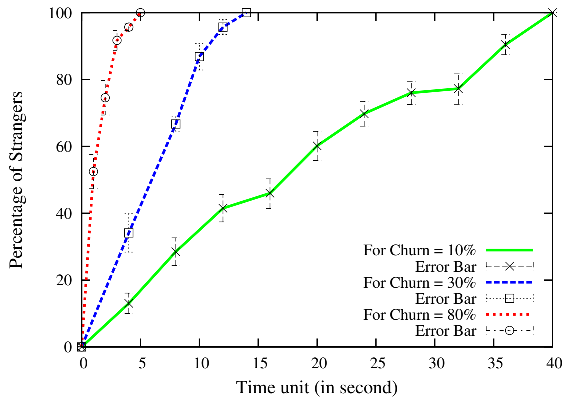


Figure 8.1 – Evaluation of Stranger model for 10%, 30% and 80% of churn

per time unit (seconds). If we assume equal probability of join/leave event and a single event per time unit, then every other time unit, a node will leave and a new node will join the network, i.e., every other time unit the total number of peers will be the same, whereas only a single node has a changed identity. So, during network partition, every other time unit, a new node replaces an existing node in a partition, about which no other node of any other partition has any knowledge. We can say that this new node becomes a stranger for the nodes on other partition. In our model, we assume uniform distribution of lifetime of peers. However, in real systems, this is necessarily not the case. So, our model is a pessimistic one, in other terms corresponds to the "worst case" scenarios. Investigation using a more realistic up-time distribution of peers is left as future work.

Suppose, the number of nodes on a SON is N. For simplicity we will consider only 2 partitions of almost equal size, to present our model, can be generalized for N partitions as well. After a network partition, two independent overlays, P_1 and P_2 are formed having n_1 and n_2 nodes respectively on each overlay, i.e., $n_1 + n_2 = N$, $n_1 \approx n_2$ and P_i is the set of nodes of partition i. The best possible starting state for a partition of the overlay is, when each partition has complete knowledge about the other, i.e., $(\bigcup_{p_i \in P_1} KB_{p_i}) \bigcap P_2 = P_2$. We present our model to generalize the interaction between network partition and churn, during the partition period, based on this simplifying assumption. For a churn intensity of $C\%$, after 1 time unit, the number of nodes on P_2 known to P_1 is, $(\bigcup_{p_i \in P_1} KB_{p_i}) \bigcap P_2 = n_2 \times \mathrm{e}^{-\frac{C}{100}}$ After t time units, $(\bigcup_{p_i \in P_1} KB_{p_i}) \bigcap P_2 = n_2 \times \mathrm{e}^{-\frac{Ct}{100}}$.

Prediction of Irreversibility: Using the stranger model, we can predict the limit (in time unit) of achieving reversibility, i.e., the number of time units, since a network partition, beyond which the system is unable to achieve reversibility by itself. For every principle, healing by merging of overlays is based on the known references of peers on the other partition, with which communication can be established as the partition ceases. So, with the increment of strangers on both P_1 and P_2, the merging becomes more difficult and at time unit T_{CO}, $(\bigcup_{p_i \in P_1} KB_{p_i}) \bigcap P_2 = \emptyset$ (and vice versa). After T_{CO} time units P_1 and P_2 will be complete strangers to each other and as the nodes on a partition have no reference

Churn (C)	Theoretical *Cut-off* Time	Measured *Cut-off* Time
10%	39.12	40
30%	13.04	14
80%	4.89	5

Table 8.1 – *Cut-off* time (T_{CO}) for different values of Churn

about any peer on the other partition, no healing mechanism of the system will be effective. We will refer to T_{CO} as the *cut-off* point, beyond which, a mechanism outside the system or third party intervention is required to make the system reversible. We can derive T_{CO} as a function of C and n_2 using our model:

$$n_2 \times \mathrm{e}^{-\frac{CT_{CO}}{100}} = 1 \therefore T_{CO} = \frac{100 \times \ln n_2}{C} \tag{8.1}$$

Using Equation 8.1, we can derive T_{CO} for P_2, with respect to P_1 for a given C. We present an experimental validation of our model for churn of 10%, 30% and 80%. For these experiments, we use a SON of 100 peers. To simulate the underlying network, the end-to-end delays are set based on the empirical distribution of minimum RTT provided in [87]. During steady state, a network partition is simulated by creating 2 partitions of the overlay of equal size, i.e., $|P_1| = |P_2| = 50$. We have verified that after the partition, i.e., at $t = 0$, $(\bigcup_{p_i \in P_1} KB_{p_i}) \bigcap P_2 = P_2$ and vice versa, for these experiments. Churn of particular intensity is injected for t seconds. To inject a churn event, a partition is chosen with equal probability. We have used correction-on-change, periodic stabilization and the merger with passive KB for maintenance to have the best measurement of strangers. After withdrawing churn, we retrieve the knowledge base of each node of P_1 and make a superset of those. We count the nodes in P_2 which are not in that set and calculate the percentage, this is the percentage of nodes in P_2, which are stranger to P_1. We do the same for P_1 with respect to P_2 and report the average for a single run. An average of 10 such runs is reported in Figure 8.1 with increasing time. We present the values of T_{CO} for 10%, 30% and 80% of churn and $n_1 = n_2 = 50$ using Equation 8.1 and from experiments in Table 8.1. As shown in Figure 8.1, the percentage of strangers increases with time, for all values of churn. Also, the *Cut-off* points coincide with those derived using Equation 8.1, as presented in Table 8.1, thus validating our model. We have expressed T_{CO} assuming the best possible starting state for a partition of the overlay, i.e., each partition has converged knowledge about the other. A generic expression for T_{CO} is subject to future work.

8.2 Evaluation of Maintenance Principles

We evaluate maintenance strategies in terms of their abilities to overcome the challenges posed by strangers, while merging multiple overlays. The generalized effect of churn of different intensities during network partition is the rate of increasing the number of strangers. So, in these experiments, we have used only one value of churn, namely 10% of churn for different durations to create desired percentages of strangers in the system. We use similar experimental setup as described in Section 8.1. We have continued the injection of 10% churn for 8, 20, 32 and 36 seconds, since the network partition is introduced. Then we withdraw churn, wait for 30 seconds for the healing of the effect of churn on both partitions and restore the network partition. We observe the reversibility of different maintenance mechanisms with time, using the same metric as in Section 7.2.2: number of islands. We present the average of 10 sample runs in Figure 8.2; however, we have excluded samples (especially for experiments with 36 seconds) for which the partitions become complete strangers (i.e., $(\bigcup_{p_i \in P_1} KB_{p_i}) \bigcap P_2 = \emptyset$ and vice versa). For stranger measurements, we have constructed a passive KB at each node for these experiments, but the KB is not used for any correction/healing (i.e., to trigger the merger of ReCircle).

8.2.1 Correction-on-*

We can see in Figure 8.2a, correction-on-* principles fail to merge the overlays even with the lowest number of strangers in the system. Also, as the duration of churn increases i.e. increment of strangers, so is the number of partitions, which remains the same throughout an experiment, after the initial healing during first 3 seconds. This is due to the lack of liveness property of these maintenance mechanisms that self-healing is discontinued.

8.2.2 Correction-on-* and Periodic Stabilization

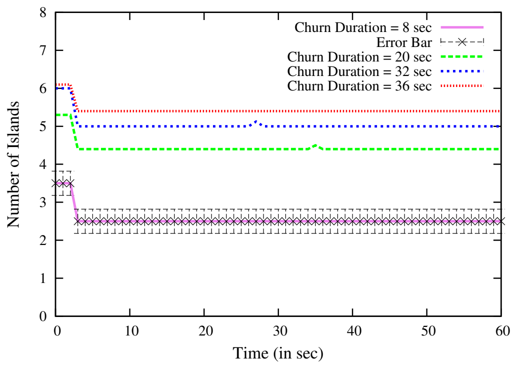
After integration of periodic stabilization, we can see significant improvement, as apparent in Figure 8.2b. The correction-on-* mechanisms, along with periodic stabilization shows reversibility, even for churn duration of 32 seconds. However, beyond that, this combined strategy fails to guarantee reversibility, i.e., for 36 seconds of churn duration, there are runs, for which merging has not converged. We have observed till 120 seconds for these runs, but the result remains the same throughout, so for presentation purpose till 60 seconds is shown.

8.2.3 ReCircle (Periodic Stabilization and Merger with passive list)

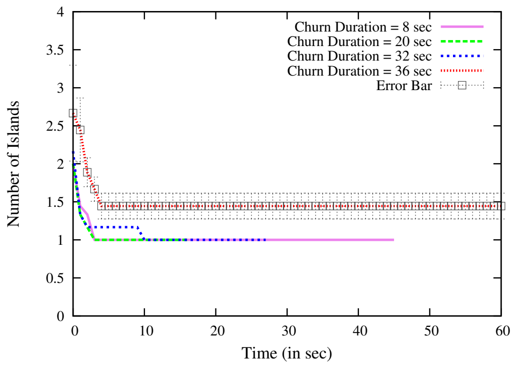
As shown in Figure 8.2c, ReCircle gives no more improvement over combined local maintenance (Figure 8.2b). It shows reversibility up to churn duration of 32 second. However, for the experiment with churn duration of 36 second, there are runs, which have not converged. We have observed till 120 second, the result remains the same, so for presentation purpose, till 60 second is shown. The reason is the lack of knowledge to trigger the merger using the passive list.

8.2.4 Knowledge Base

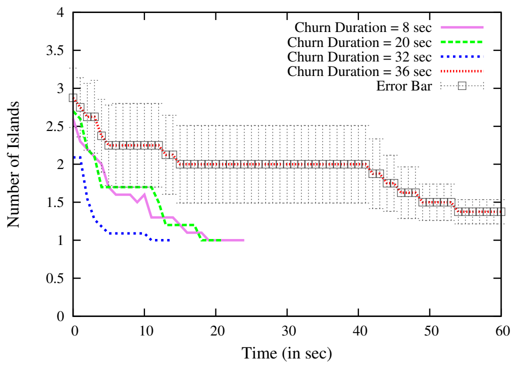
ReCircle (Periodic Stabilization and Merger with passive list) works well for up to 32 second churn, i.e., 77% strangers (see Figure 8.1), whereas adding a passive Knowledge Base works up to 36 second churn, i.e., 90% strangers (and beyond, verification of which is subject to future work), as shown in Figure 8.3a. This is much closer to the limit of 100% strangers that is reached at 40 second churn. This clearly shows the effectiveness



(a) Using Correction-on-*

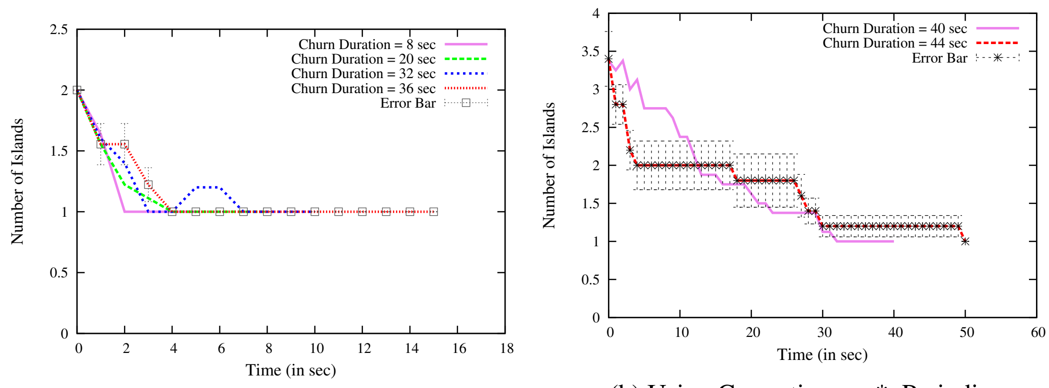


(b) Using Correction-on-* and Periodic Stabilization



(c) Using Periodic Stabilization and the Merger with passive list

Figure 8.2 – Number of islands as a function of time (in second) after withdrawing churn and partition to assess self-healing against strangers using different maintenance strategies



(a) Using Correction-on-*, Periodic Stabilization, and the Merger with Knowledge Base

(b) Using Correction-on-*, Periodic Stabilization, and the Merger with Knowledge Base and Oracle

Figure 8.3 – Number of islands as a function of time (in second) after withdrawing churn and partition to assess self-healing against strangers using different maintenance strategies

of the Knowledge Base. We have used $\sigma = 1$ second to optimistically use elements of the knowledge base to trigger the merger. Also, we have integrated the correction-on-change principle (Section 3.3.1) to avoid any inconsistency while handling churn using periodic stabilization as the only local correction policy (as discussed in Section 3.3.3), without causing any extra load on bandwidth consumption. Using all three mechanisms (Correction-on-*, ReCircle, and Knowledge Base) gives fast convergence of number of islands to 1. Using only two of these three mechanisms will either not converge to 1 or else converge much slower to 1. We have excluded samples for Figure 8.3a, for which the partitions become complete strangers (i.e., $(\bigcup_{p_i \in P_1} KB_{p_i}) \bigcap P_2 = \emptyset$ and vice versa). Because for these scenarios, this combined healing falls short, as there is no knowledge in a partition about the other to achieve reversibility.

8.2.5 Oracle

As we can see in Figure 8.3b, when the partitions become complete strangers to each other, it is possible to achieve reversibility, as long as an oracle injects the lost knowledge about the other partition. We have implemented an oracle in the application layer, which every 5-second, picks up 2 pairs of nodes from its list and introduces them to each other through an API. The list of peers at application layer can be built in either a active or passive way. As per the active approach, the application layer can periodically retrieve knowledge base from the peers it knows, thus building a superset of these knowledge bases, whereas in the passive approach the application layer comes to know about peers while joining. In our experiments, we have used the second approach to build the list at the application layer. Figure 8.3b shows the convergence for ≥ 40 seconds churn, i.e., 100% strangers (see Figure 8.1). Also, for these experiments, we have restored network partition immediately, instead of waiting for 30 seconds, in order to observe the overall impact the oracle has on

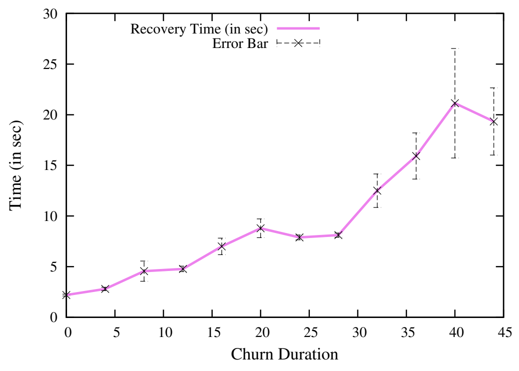


Figure 8.4 – Recovery/Healing time for increasing strangers

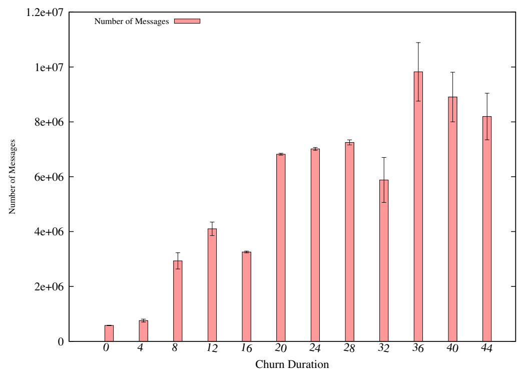


Figure 8.5 – Number of messages generated for increasing strangers

the healing process. For this reason, in the first couple of snapshots the number of islands is higher than 2, as a result of the temporary inaccuracy in the neighborhood of each node, due to churn.

8.3 Recovery Time and Cost

We present recovery time and cost, in term of number of messages, to achieve reversibility against increasing strangers among the partitions of the system. We have used similar experimental setups and induced churn of 10% for increasing duration during the partition. Correction-on-* and ReCircle with the knowledge base approach (passive KB and oracle) are used as part of maintenance for these experiments. After withdrawing churn, partition is restored and we measure the time (in seconds) required for complete healing, i.e., all nodes organized into a perfect ring topology. We have used a mean value of 20 independent runs for every 4 seconds increase of churn duration. Figure 8.4 shows the result. This along with Figure 8.1 gives an idea regarding the percentage of strangers in the system and the time required to achieve reversibility. As expected, the recovery time increases with the number of strangers in the system. We report the average number of messages generated for increasing churn duration in Figure 8.5. This also follows the similar pattern as in Figure 8.4. We notice large number of messages generated for a network of 100 peers, raising concern about scalability of the system. However, by controlling the number of messages triggered by the merger of ReCircle using the knowledge base and oracle parameters and setting an optimistic period for periodic stabilization, the bandwidth consumption can be lowered, while trading-off convergence time. Also, it is essential that the system should adapt with the operating environment, which we keep as future work.

8.4 Related Work

Several works are done to assess resilience of various maintenance strategies under churn [103], [34], [104], [31]. However, we have not found any work that analyzes the impact of the interaction of network partition and churn, thus to quantify the challenges posed by this interaction on the maintenance strategy of the overlay. In our work, we have presented a model, using which it is possible to generalize the effect of network partition under churn of any intensity and duration. Also, this can provide useful information to the application layer regarding the cut-off point, beyond which the application or third-party intervention is required to take initiative by injecting knowledge, in order to achieve reversibility.

8.5 Discussion

In this chapter, we have investigated the interaction between *Network Partitioning* and *Churn (node turnover)* in *Structured Overlay Networks*. We have proposed a model, namely "Stranger Model", to generalize the impact of *simultaneous* network partition and churn. We have shown that this interaction causes partitions to eventually become strangers to each other. Suppose, S_{old} and S_{new} are the global states of the system before the system faces network partition and after the partitioning ceases respectively, where S_{old} and S_{new} are the sets of local states of all valid nodes of the system at those instances. The partitions become complete strangers to each other when $S_{old} \cap S_{new} = \emptyset$, which makes full reversibility impossible when this happens. Using stranger model, we have also identified the boundary (cut-off point), beyond which the system is unable to achieve reversibility by itself. We propose to use knowledge base based resilient maintenance, i.e., unrestricted gossiping to handle partitions under churn, that contains knowledge collected in a passive way at each node and injected by an oracle beyond the cut-off point. We conclude that as the global state of the system is perturbed due to churn during network partitioning, the strongest form of resilient maintenance has no alternative to ensure reversibility of the system.

Chapter 9

Packet Loss

In this chapter, we consider only one stress element, namely *Network Dynamicity* due to packet loss in physical network. Packet loss on a network occurs when packets arrive at a router or network segment at a higher rate than the physical limits of these equipments. As discussed in the literature [83, 105, 106, 107], packet loss on the Internet is bursty in nature, i.e., if the previous packet is lost then there is higher probability that current packet will also be lost. Also, for a system running on an edge network, like our representative system, packet loss occurs at different location: (i) packet loss due to edge congestion: queue overflows at ISP edges, (ii) packet loss due to core congestion: queue overflows at routers of core IP network. In this chapter, by means of simulations, we demonstrate reversibility, once the bursts of packet loss ceases, for overlay networks with high rates of packet loss and we make general conclusions about the ability of the maintenance strategies to achieve reversibility for the system. While simulating end-to-end packet loss model, for simplicity purpose, we have ignored TCP congestion avoidance measures. We have made such simplifying assumption due to the inherent complexity of TCP and its environment, also the assumption is justifiable following the existing literature. In other words, we say that we have simulated an UDP packet loss model.

9.1 Evaluation of Reversibility

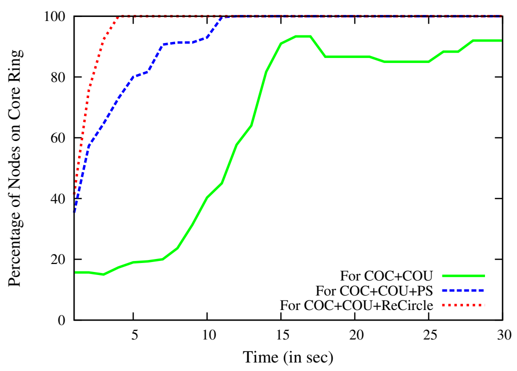
Local Maintenance (combined reactive and proactive corrections) is sufficient for Reversibility. After suffering from a burst of packet loss due to congestion in the underlying physical network, the system should reverse back to original state, i.e., all nodes should eventually converge to a perfect ring. In this section, we assess reversibility of each maintenance strategy and also find the limit (if any), after a burst of packet-loss ceases. Our experimental results, presented in Figure 9.1, verify that local maintenance is sufficient to achieve reversibility against such stressful operating environment caused by packet-loss in the physical network; though the local maintenance needs to be both proactive and reactive.

In our evaluations, we use our representative system, Beernet [13]. We assess reversibility in stepwise fashion, by integrating a new maintenance principle at each step and evaluating the behavior of the resulting system. For assessment of reversibility we use a SON of 100 peers. All experiments are done in Mozart-Oz 2.0 [95] in a simulated environment. To simulate the underlying network, the end-to-end delays are set based on the empirical distribution of minimum RTT provided in [87]. The distribution represents significant geographic diversity, so we can say that the simulated SON is geo-distributed. We have simulated packet-loss following drop-tail loss model [83], as described in Section 4.1: each node p is assigned a maximum Bandwidth, $BW(p)$, in terms of number of messages per time unit. We have assumed each message is equivalent to one packet on the physical network. Such assumption is reasonable, as a message in our system is a record, where fields of the record are basic data types (e.g., integer, float, atom, list etc.). So, each node p is able to send maximum $BW(p)$ messages per time unit. During each time unit (second in our experiments), messages are considered sequentially. For each message with a sequence number of n during a second , if $n < BW(p)$, the message is lost with a probability of $\log_{BW(p)}(n)$ (corresponds to packet-loss due to core-congestion), otherwise, the message is lost with probability 1 (packet loss due to edge congestion). The maximum BW for nodes are chosen from a range: $[a, b]$, where $a \leq b$. The maximum BW of nodes are uniformly distributed in this range. We have used a range in order to control the heterogeneity in the system. If $a == b$ then all nodes in the system are assigned the same maximum BW. We have varied the maximum BW range in order to create stressful heterogeneous operating environment for the system. We have used four manimum BW ranges in our experiments.

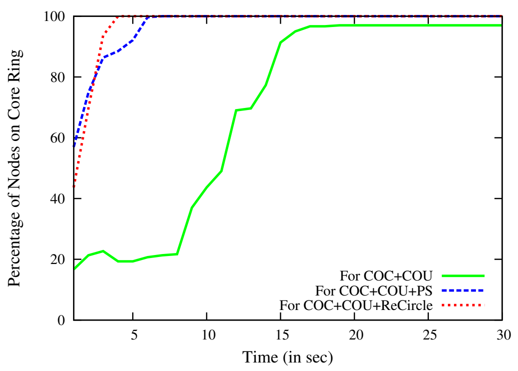
- Homogeneous Environment: $[50, 50]$ (50 messages per second as the BW for all nodes in the system),

- Heterogeneous Environment: $[50, 80]$, $[50, 100]$ and $[50, 160]$ (the BW of nodes, in terms of number of messages per second, are uniformly distributed in these ranges)

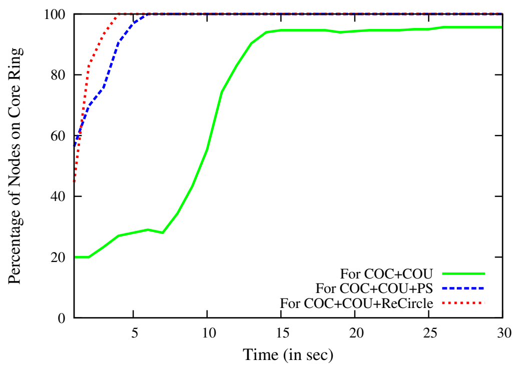
We have used four increasingly wider ranges in order to compare and study how heterogeneity impacts Reversibility. We have simulated burst of packet loss for 2 minutes as per the loss model described above. We have chosen the duration of stress as two minutes because as per [108], over 70% of Internet failures have durations less than two minutes. After withdrawing packet-loss (in order to verify weak reversibility of the system), we observe healing capability of SON with time. In order to quantify the effect of self-healing, we have used the following metric: percentage ($\%$) of nodes on core ring, to measure the rigidity of the ring. In other terms, we find out the maximal ring in the system and report percentage of nodes on it. The ultimate goal is to have the metric converge to 100%. A fixed workload is used in an experiment by injecting transactions, modeled as an HPP with $\lambda = 1$ transaction per second. A transaction reads one key and updates another one. Since the withdrawal of churn, for each second we present percentage of nodes on core ring and an average of 10 independent runs are taken for each second. We remark that the termination time of an experiment in Figure 9.1, apparent in the result, is the maximum among the samples used.



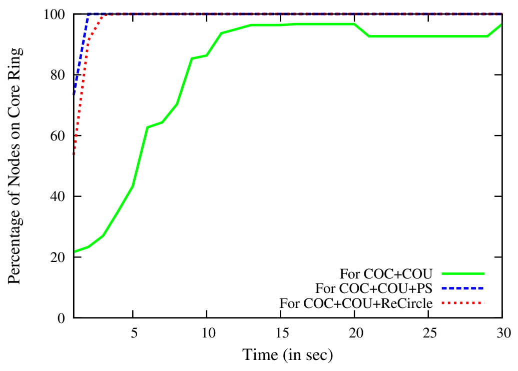
(a) Max BW Range $[50, 50]$



(b) Max BW Range $[50, 80]$



(c) Max BW Range $[50, 100]$



(d) Max. BW Range $[50, 160]$

Figure 9.1 – Percentage of nodes on the core ring as a function of time (in second) after withdrawing burst of message loss to assess reversibility. Using only reactive local maintenance is unable to achieve reversibility for the system. Combined local maintenance: reactive Correction-on-* and proactive periodic stabilization, is sufficient to make the system reversible. Integration of a global reactive maintenance, merger with passive list, does not achieve any significant improvement when there are nodes with high BW in the system; however when all nodes in the system have the same low BW (Figure 9.1a), recovery time of the system is reduced by more than 50%.

9.1.1 Correction-on-Change and Correction-on-Use (Correction-on-*)

Correction-on-*, due to its rapid reaction against an event (join/leave/failure/false suspicion) exhibits certain healing capabilities, however, fails to complete the healing process. We can see in Figure 9.1, irrespective of the maximum BW range, there are some runs for which correction-on-* has failed to establish weak reversibility of the system as the period of packet loss ceases. We have observed for 3 after withdrawing packet-loss, the results remain the same for all four. The reason is that, due to packet loss each node falsely suspects others in the system. As a result of such false suspicions in the system, some nodes are pushed on the branches, where some others are partitioned away and ringlets are formed. However, each node continues monitoring those suspected peers. As no churn is experienced, so the list of monitored nodes is not altered. So when it can make a connection with those suspected peers, a false suspicion event is triggered, as a result of which it corrects its pointers, resulting in merging of branches and ringlets to the core ring. However, as we can see in Figure 9.1, there are runs for which correction-on-* as the sole maintenance has failed to complete the recovery process. The reason is the lack of liveness of this maintenance: as the events (detection of false suspicions in this case) fade away, healing is also discontinued, resulting in few branches to remain in the system.

9.1.2 Correction-on-* and Periodic Stabilization

After integration of Periodic Stabilization, the system is reversible for all four experiments we have conducted. The period used in our experiments is 3 seconds. Each node independently performs periodic maintenance by exchanging messages with its successor to maintain the local geometry. Due to such periodic maintenance, nodes on branches eventually become part of the core ring. As we can see in Figure 9.1, the combination of reactive and proactive local maintenance is sufficient to ensure weak reversibility of the system against stressful operating environments due to packet-loss. As we can see in Figure 9.1a, it takes more time to converge to a perfect ring when the nodes has the same low BW of 50 messages per second, thus suffer higher rate of packet-loss. As a result the embedded graph of the SON is damaged more, requires more time to complete the healing. However, as expected when there are nodes in the system with higher BW (e.g., Figure 9.1b,9.1c,9.1d), we can see significant improvement in the recovery time, especially in Figure 9.1d.

9.1.3 Correction-on-* and ReCircle (Periodic Stabilization and Merger with Passive List)

As Figure 9.1 shows, the integration of merger with passive list does not show much improvement over the combined local healing, where are nodes with higher BW exist in the system. The period used to dequeue the elements to generate *mlookups* at each node is 5 seconds and we have kept the *fanout* parameter as 1. As evident in Figure 9.1a, when the system suffers higher rate of packet-loss, leading to intense topological damage, a reactive merger reduces recovery time significantly. The reason is that as the false suspicions are detected, the passive list populates the queue of the merger at each node. The merger using its gossip algorithm periodically spreads the maintenance further in the identifier space, triggering local maintenance during subsequent rounds of periodic stabilization. As a result the convergence to a perfect ring is much faster. However, when there are nodes in the system with higher BW, the topological damage is much less due to packet-loss, so local maintenance is sufficient to recover efficiently. Thus, no significant improvement is visible in Figure 9.1b,9.1c,9.1d after integration of merger with passive list.

9.2 Related Work

In this section we discuss relevant works on structured overlay network experiencing packet-loss in the underlay network. In [29], an evaluation of dependability and performance of

MSPastry (a new implementation of Pastry[96]) is conducted by varying link loss rates, where they have used Bernoulli loss model [83] in their simulator. As they have observed, even with high link loss rate (network loss rate is varied between 0% to 5%), the system continues to provide reasonable performance and dependable routing with high probability, though there exists a small probability of inconsistency. There are several works in the literature which validates the performance and reliability of a particular application running on top of a structured overlay network, in lossy operating environments. In [43] structured overlay networks have been proposed to be used as resilient routing infrastructures. By simulation they investigate the impact of random link failures on Tapestry [109] in the wide area. They incrementally inject randomly placed link errors into the network and observe connectivity of pair-wise paths between all overlay nodes. Their results show that for the large majority of paths where connectivity is maintained after failures, routing is successful. [44] presents a packet loss and overlay size aware broadcast algorithm for Kademlia [110] overlay network and assesses reliability of their algorithm with a packet loss ratio of 20%, i.e., they have used Bernoulli loss model with $p = 0.2$. The work in [111] outlines and evaluates the practical way to build a structured overlay network to deliver emergency traffic over the Internet. They discuss mechanisms of how overlay nodes can deal with and recover from packet-loss on the Internet and ensure reliable delivery of emergency traffic. They categorize reliabilities into node-by-node and end-to-end reliability, where for node-by-node, each intermediate overlay node deploys a reliable protocol, and keeps track of and deals with packet-loss individually. For end-to-end reliability, end-to-end acknowledgements are exchanged between end receiver and initial sender. In our system, node-by-node model is adapted, where each node deals with packet-loss on individual basis and does maintenance accordingly. Our work on evaluation of reversibility of a structured overlay network facing packet-loss can be seen as complementary of these works.

9.3 Discussion

In this chapter, we have investigated network dynamicity due to packet loss in the physical network. We have considered scenarios, where no churn is experienced by the overlay during such dynamicity in the physical network, which correspond to bursts of packet-loss of short durations. We have presented a comparative analysis of weak reversibility of existing maintenance strategies, namely, correction-on-*, periodic stabilization and ReCircle (as done in [37]). Our results show that when the global state of the system is not perturbed, i.e., the system does not experience any churn during the packet-loss (though such communication problems might cause individual nodes to change their local states, e.g., change of successor and/or predecessor pointers), micro-level interactions among nodes (i.e., local corrections at each node) are able to trigger and accomplish macroscopic healing (i.e., convergence to a perfect ring overlay). However, the local corrections need to be both proactive and reactive to ensure complete recovery. Also, we have observed that the integration of a more resilient stronger algorithm, e.g., gossiping, reduces the recovery time significantly in case where each node in the system experiences higher rate of packet-loss;

though in heterogeneous environments, where some nodes have higher BW than the others, the improvement is minimal.

Part IV

Practical Applications

Chapter 10

Efficient Approximation of System Functionality

A *Structured Overlay Network (SON)* provides significant functionality to the applications running on top, e.g., transactions over key/value store. However, as the in-hospitality of the operating environment of a SON continues to increase, it will no longer be able to provide such functionalities. Thus, applications that rely on transactions will no longer be able to use them. We would like these applications to continue running nevertheless, with predictable behavior even though functionality will be less. Ideally, this should be done in a manner that works even for operating environments which are extremely inhospitable. The SON can therefore not be relied on to do additional computation to determine its level of functionality. Under this constraint, is it possible for the SON to give useful information to the application?

As discussed in Chapter 2, the *reversibility function* ($F_{rev}(S(t))$) computes the set of available operations (i.e., the functionality of the system) of a reversible system at any instant. An operation is *available* for a given stress if the operation will eventually succeed (i.e., it will fail only a finite number of times if tried repeatedly, and then succeed). As we have defined, the reversibility function depends only on the current stress in the operating environment of the system. However, stress is a global condition that cannot easily be measured by individual nodes. So, in order to approximate the reversibility function we have introduced the concept of *Phase*. Unlike stress, phase is a per-node property. All nodes in the same phase exhibit the same qualitative properties, which are different for nodes in different phases. We have defined the *Phase Configuration* of the system to be the vector $P_c = (P_1, P_2, P_3, \ldots, P_n)$. We have proposed $F_{phase}(P_c(t))$, the *phase function* to approximate the reversibility function, i.e., to approximate the available functionality of the system. In this chapter we have applied these concepts to approximate the reversibility function, i.e., to compute the phase function for our representative system. We investigate phases and phase transitions in our representative system due to stress. We experimentally demonstrate *Reversible Phase Transitions*: the nodes of the system change phase as the stress varies. We note that the concept of phase and phase transition are analogous to phase in physical systems; we have chosen the term phase for this very reason. Phase transitions is the consequence of having a reversible system, i.e., the system we have constructed is able to survive extremely high levels of stress.

10.1 Approximating the Reversibility Function

In this thesis we propose an efficient approximation approach to estimate the reversibility function, i.e., to compute the phase function for our representative system. We do this based on the assumption that the rest of the system is qualitatively the same as the current node, i.e., at node i, $P_c[j : j \neq i](t) = P_i(t)$. Thus, each node i computes the phase function, thus approximates Op_{avail}, the set of available operations of the system at any instant, based on its current phase. The reason behind such simplification is to avoid any extra distributed operation or bandwidth consumption. In order to have a better approximation, each node needs to maintain a *Phase Base (PB)* (a local view of P_c) and communicate with others to enhance its PB. This can be done by piggybacking PB with messages routed via each node, to avoid any extra distributed operation; however this still consumes extra bandwidth. The implementation of PB, and the investigation about extra resource consumption, to justify the worthiness of such protocol, are left as future work.

10.2 Phases in Relaxed Ring

In this section we explain how we apply the definition of phase (see Section 2.2) to identify different phases, sub- phases in our representative system, Beernet [50]. Next, we describe semantics of each phase and sub-phase in terms of available operations, i.e., we compute the phase function based on our simplifying assumption described above.

In the case of our representative system, Beernet, we can determine a property that satisfies the formal definition of phase. The SON's phase is a qualitative description of the structure of the SON. For example, at low churn, the SON has a mostly fixed structure, and at high churn, the SON can decompose into small rings or single nodes. For low churn, the SON has full functionality, and at high churn, the SON has reduced functionality. The phase of the SON is not a global property, but is observed separately at each node, and can be different for different nodes. No global synchronization and no extra computation is required to compute the phase; it is a direct consequence of the observed structure of the SON at each node. Thus, the phase inferred at each node correlates with SON functionality and can allow the application running on that node to modify its behavior depending on the functionality available for the current stress.

The phase of a node in our representative system, Beernet, is clearly determinable at that node: there are three mutually exclusive situations depending on neighbor behavior (neighbors on core ring, neighbors on branch, no neighbors). As we will see, there is a close analogy between these three phases and the solid, liquid, and gaseous phases in physical matter (e.g., water). Also, when a node is on a branch (i.e., liquid phase), we can identify three sub-phases in terms of available functionalities and probability of facing an immediate phase transition. All these phases and sub-phases are distinguishable from each other and provide sufficient information to understand the available operations of the system. Each node changes its phase independently. The current phase and phase transition at each node can be determined with high confidence, without any global synchronization (will be discussed in Chapter 11). However, at system level, phase transitions happens when there are changes in the phases of the majority of nodes. When external conditions change, each node changes phase. If that happens to many nodes, we have a phase transition at system level. It is not possible to determine at a single node, a phase transition at system level, because this would require a global algorithm.

The phase of each node has a direct co-relation with the overall properties (e.g., routing, availability of keys) of the system. The routing guarantee offered by the system is $O(\log_k(N) + b)$, where N is the total number of nodes and b is the branch size (i.e., the average number of nodes on branches) of the system. So, the more nodes change to liquid phase, the routing will degrade with that. Also, with the increased branches in the system, the probability of introducing unavailability of keys under stress, e.g., churn, also increases due to isolation of branches (as discussed in [13]). We define semantics of each phase and sub-phase, in analogy with the solid, liquid, and gaseous phases in physical matters [52]. This analogy is introduced to make it easier to understand intuitively what each phase means. We define the semantics of each phase and sub-phase, in the context of Op_{total} (see Section 3.2), the functionality set of our representative system.

- **Solid (P_S):** The *solid* state of a matter is characterized by structural rigidity, where atoms or molecules are bound to each other in a fixed structure. In case of SONs, if a peer has stable predecessor and successor pointers (i.e., the peer is on core ring), along with a stable finger table, it can be termed to be in solid phase. It can be safely assumed that such peer can support efficient routing, thus accommodate up-to-date replica sets, thus leading to all the upper layer functionalities, e.g., transactional DHT.

- **Liquid:** A thermodynamic system is in the liquid state when molecules are bound tightly but not rigidly. In case of a SON, if the peer is on a branch, it is less strongly connected to the system than the nodes which are in *solid* phase. However a peer can be on a branch temporarily, e.g., as part of the join protocol or due to false suspicion as a result of sudden slow-down of underlying physical link. We identify three liquid sub-phases.

  - *liquid-1 (P_{L1})* If the peer is on a branch, but the depth of the peer (distance from the core ring) is less than or equal to 2. Also, the peer still holds a stable finger table. The justification of depth of 2 for this sub-phase is based on the evaluation of average branch sizes in [13], where it is shown that the average size of branches of Beernet is ≤ 2, corresponding to the connectivity among peers on the Internet. So, if a peer's depth on a branch is ≤ 2, the operating environment from a peer's perspective is still the usual one, it might temporarily be pushed on a branch. From the application's perspective, the peer is still able to provide transactional functionality to the application.
  - *liquid-2 (P_{L2})* If the peer is on a branch, but the depth of the peer (distance from the core ring) is greater than 2 and it is not the tail of the branch. Also, the

finger table at the peer still holds $> 50\%$ valid fingers. So, the peer is still able to support at least all DHT operations, however successful transactions are not guaranteed anymore.

  – *liquid-3 (P_{L3})* If the peer is on a branch with a depth > 2 and it is the tail of a branch. As discussed in [13], the tail of a branch has higher probability to get isolated during churn, thus introducing unavailability in the key range. Also, most of the fingers in the peer's finger table are invalid or crashed. From the application perspective, the peer in this sub-phase provides very limited functionality, mostly basic connectivity through its successor pointer.

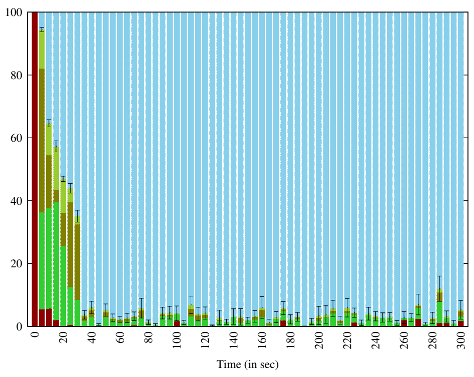
- **Gaseous (P_G):** The gaseous state of matter is made up of individual molecules that are separated from each other. When Beernet experiences high stress, nodes are partitioned-away, thus becoming isolated. Such isolated nodes, which do not have connection with any other nodes of the system, can be termed to be in gaseous phase. As shown in Figure 3.2, ringlets can be formed due to stress in the operating environment of the system. For example, if there are > 1 nodes per physical machine and the physical network breaks down, then ringlets will be formed by the nodes on the same physical machine. For such scenarios, using the global view we can term the nodes on a ringlet to be in a form of gaseous sub-phase. However, as phase is a node specific local property, a node on a ringlet determines its phase based on its local view, thus its phase decision will be either *solid* or a *liquid* sub-phase.

10.3 Observation of Phase Transitions

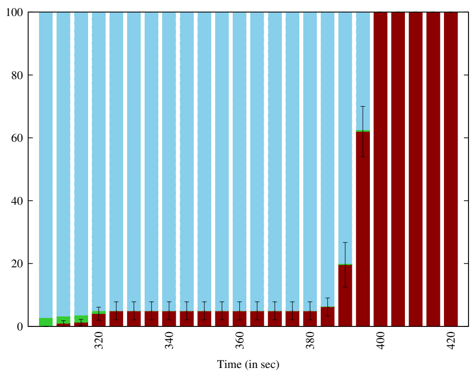
We empirically demonstrate reversible phase transitions in a reversible system [50]. For our experimental study, we have used a network size of 1024 peers. To simulate the underlying network, the end-to-end delays are set based on the empirical distribution of minimum RTT provided in [87]. We have used both efficient and resilient maintenance during these experiments in order to study and identify the optimal critical points. A fixed workload is used in an experiment by injecting transactions (using paxos consensus [112] based commit protocol [94]), modeled as an HPP with $\lambda = 1$ transaction per second. A transaction reads one key and updates another one. The system uses *Symmetric* Replication [70], with a replication factor of 4. In this chapter, we have used two stress elements, churn and network dynamicity due to packet-loss, to cause the system's operating environment to be inhospitable. We measure the percentages (%) of nodes in different phases and sub-phases.

10.3.1 Churn as the Stress

In this section, we consider the environment in-hospitality to be measured by the *Churn* parameter, i.e., the rate of node turnover (nodes failing and being replaced by new correct nodes). We show experimentally the existence of phase transitions in our representative reversible system as the churn intensity varies.



(a) Under increasing churn



(b) After withdrawing churn

Figure 10.1 – Phase Transitions in Beernet: red, green (different shades corresponds to 3 liquid sub-phases) and blue (dark, gray and light-gray in B/W) areas correspond to percentages of nodes in solid, liquid and gaseous phases respectively

Increasing Churn with Time

In this experiment we study phase transitions in our representative system under increasing churn and reverse transitions when churn is removed. We start with 5% churn and increase the intensity by 5% every 5-second for 5 minutes. As we reach churn value of 100, we keep on injecting 100% churn for the rest of the period. Every 5-second we take a snapshot of the system, i.e., the percentages of nodes at each phase and sub-phase throughout a run. After churn is withdrawn, we let the system run until the completion of self-healing (i.e., perfect ring) and every-5 second take the measurements. We have used mean value for 20 independent runs. Figure 10.1a and Figure 10.1b show the states of the system during increasing churn followed by zero churn respectively.

Only error bars for gaseous phase (P_G) are shown in Figure 10.1a. The red area of each bar corresponds to percentage of nodes which are in solid phase (P_S). At time 0, i.e., starting of the experiment, all nodes are organized into a perfect ring. As churn is increased nodes start moving on branches, the green area of each bar, these are the nodes, which are in liquid phase. We have also identified nodes on branches which have different liquid sub-phases, as per the semantics described before. The result shown is an average of the parameter values for several samples and each sample go through a phase transition at different instant of time. However, still we can figure out some trends apparent in Figure 10.1a. For example, 30% of churn is a critical value, observed at 30 second, as a significant fraction of nodes change from liquid to gaseous phase.

The solid to liquid transition happens between 0% and 5% churn. In Figure 10.1a, we can see a sharp fall of percentage of nodes on core ring from 0 to 5 second. In order to analyze this transition we have zoomed into this area. For this experiment, during steady state of SON, we start with 1% of churn and every 5-second we increase churn intensity by 1% till we reach 5% of churn. Also a snapshot is taken every 5 second as before. Figure 10.2 shows the result. We have used mean values of 20 samples and only error bars

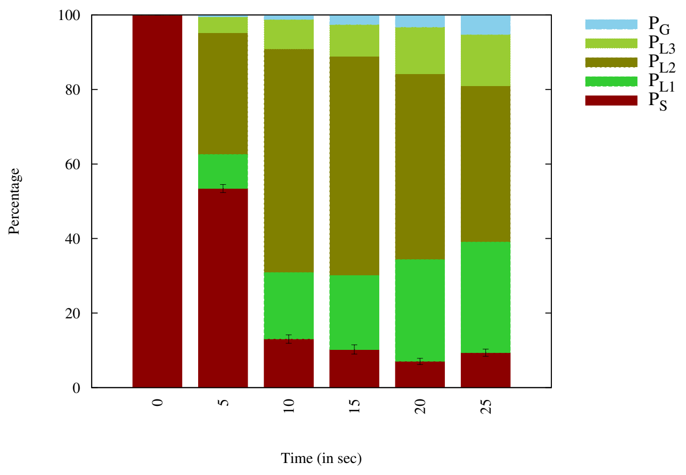


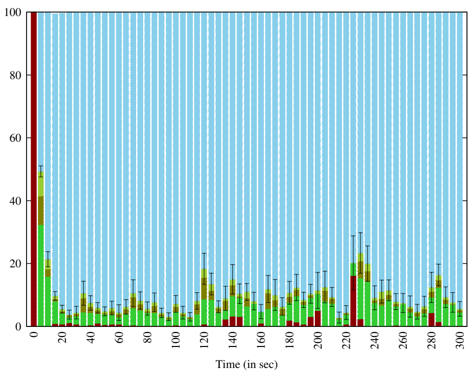
Figure 10.2 – Phase Transitions in Beernet under low churn (0% to 5%): red, green (different shades corresponds to 3 liquid sub-phases) and blue (dark, gray and light-gray in B/W) areas correspond to percentages of nodes in solid, liquid and gaseous phases respectively

for the P_S are shown. As we can see, as the churn intensity is increased from 1% to 2%, during 5 to 10 seconds, a large fraction of nodes changes phase from solid to liquid.

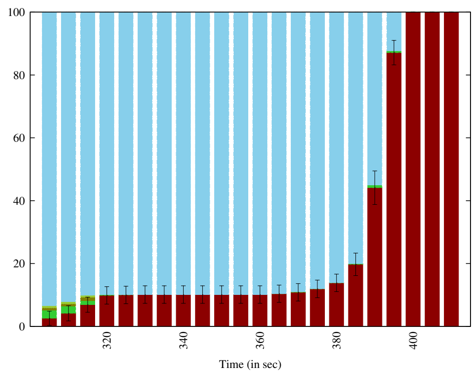
Figure 10.1b (error bars for solid phase (P_S) only) shows the recovery of the SON after churn is withdrawn until the completion of self-healing, i.e., all nodes are organized into a perfect ring (i.e., 6^{th} and 7^{th} minute of our experiment). Here starting with all isolated nodes, a small fraction of nodes changes to transient liquid phase. We can see a period of about 90 seconds, during which the percentage of isolated nodes remains same. The reason is the *Join Timeout* parameter (see Section 6.1), which is set as 90 seconds. The transition from gaseous state is controlled by this tunable parameter. Finally, all nodes are self-organized into a perfect ring, solid state of SON, within 400 seconds.

Continuous Moderate Churn

In this experiment we seek answer to the question: whether the entropy of a SON can be increased by continuous injection of low/moderate churn so that a phase transition happens in a SON. In order to investigate about this, we start with same experimental setup described before, but this time instead of increasing churn with time, we inject same value of churn. For this experiment we have chosen churn value of 30, as we have observed that 30% of churn is a critical value. During steady state of SON, we start injecting 30% churn for 5 minutes. Then churn is withdrawn and we let the SON do self-healing until all nodes are on the core ring. During our experiment, we take measurements every 5-second and present mean value of 20 independent runs in Figure 10.3a (error bars for only gaseous phase, P_G) and Figure 10.3b (error bars for only solid phase, P_S). As we can see in Figure 10.3a, a significant fraction of nodes change phase during first 10 seconds,



(a) Under continuous churn of 30%



(b) After withdrawal of continuous 30% churn

Figure 10.3 – Phase Transitions in Beernet: red, green (different shades corresponds to 3 liquid sub-phases) and blue (dark, gray and light-gray in B/W) areas correspond to percentages of nodes in solid, liquid and gaseous phases respectively

thus justifying our deduction that churn intensity of 30% to be a critical point. Also, we notice that, as in a thermodynamic system (e.g., water), it takes longer time for the system to reach a gaseous state, in fact there is no clear transition where all nodes are in gaseous phase. During injection of 30% churn for 5 minutes, a small fraction of nodes remains in solid or liquid phase surrounded by the remaining large fraction be in gaseous phase. The reverse transition shown in Figure 10.3b follows the same pattern as in Figure 10.1b.

Gradual Increase and Decrease of Churn

Till now, we have withdrawn churn completely; but what behavior the system exhibits if the intensity of churn is gradually decreased? In this experiment we investigate this. For this experiment, we use same experimental setup and parameters as already mentioned. During steady state of SON, we start injecting 5% of churn and increase the intensity of churn every 5-second until churn is of 100%. Then we gradually decrease churn by 5% every 5 second until it reaches 0. We take measurements every 5 second throughout the experiment and present mean values of 20 independent runs in Figure 10.4 (error bars for only solid phase, P_S). The behavior follows our previous deductions. Around 30% of churn a significant fraction of nodes change phase. Between 40 and $100 - 105$ seconds we see a gaseous system. During gradual decrease of churn intensity, there is increasing connectivity among nodes, followed by organization into ring structure, evidential of reversible phase transitions in our system due to increasing and decreasing churn. This provides experimental justification of the conjectured phase transitions for relaxed ring SON in [113].

10.3.2 Packet Loss as the Stress

In this section, we consider the environment in-hospitality to be measured by network dynamicity due to packet-loss in the physical network. We show experimentally the existence

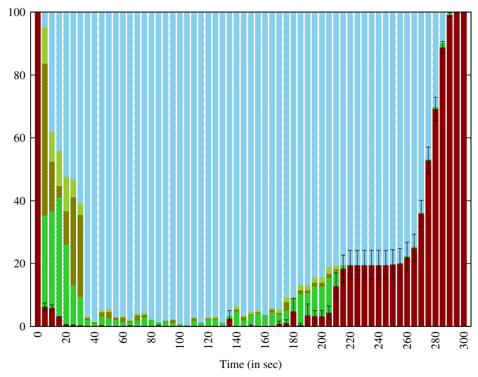


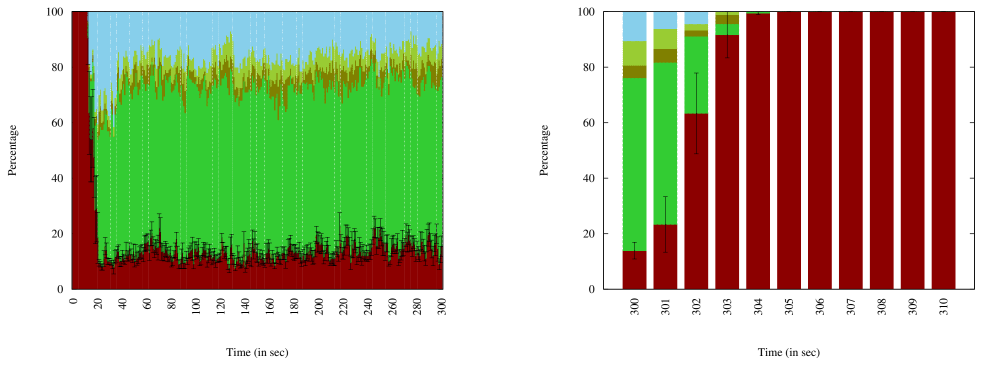
Figure 10.4 – Phase Transitions in Beernet under increasing and decreasing churn: red, green (different shades corresponds to 3 liquid sub-phases) and blue (dark, gray and light-gray in B/W) areas correspond to percentages of nodes in solid, liquid and gaseous phases respectively

of reversible phase transitions in our reversible system as the system suffers from packet-loss for a longer period.

Homogeneous Environment: The Same Low Maximum BW for Each Node

In this experiment we study phase transitions in our representative system when all nodes in the system are assigned the same low BW of 50 messages per second. We simulate our packet-loss model as described in Section 9.1 for 5 minutes: at each node, for each message with a sequence number of n during a second , if $n < 50$, the message is lost with a probability of $\log_{50}(n)$, otherwise, the message is lost with probability 1. Every second we take a snapshot of the system, i.e., the percentages of nodes at each phase and sub-phase throughout a run. After 5 minutes we revoke packet-loss, we let the system run with "no stress" until the completion of self-healing (i.e., perfect ring) and every second take the measurements. We have used mean value for 10 independent runs. Figure 10.5a and Figure 10.5b show the states of the system during packet loss followed by zero packet-loss respectively.

Only error bars for solid phase (P_S) are shown in Figure 10.5a. The red area of each bar corresponds to percentage of nodes which are in solid phase (P_S). At time 0, i.e., starting of the experiment, all nodes are organized into a perfect ring and it continues to remain the same till 10^{th} second. After first 10 seconds nodes start moving on branches, the green area of each bar, these are the nodes, which are in liquid phase. We have also identified nodes on branches which have different liquid sub-phases, as per the semantics described before. We also observe that some nodes change to gaseous phase, i.e., partitioned-away from the system. All these phase transitions at nodes happen due to false suspicions in the system as a consequence of packet-loss. Though the result shown is an average of the parameter values for several samples, some trends are explicit in Figure 10.5a. The percentage of nodes in solid phase reduces in a stepwise fashion, however during 12^{th} – 13^{th} and 17^{th} – 18^{th} second, a significant fraction of nodes ($> 20\%$) change their phase



(a) Under Packet Loss (b) After withdrawing packet-loss

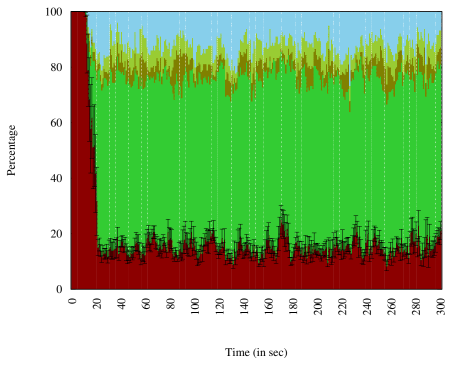
Figure 10.5 – Phase Transitions in Beernet (nodes have the same maximum BW of 50 messages per second): red, green (different shades corresponds to 3 liquid sub-phases) and blue (dark, gray and light-gray in B/W) areas correspond to percentages of nodes in solid, liquid and gaseous phases respectively

from solid to liquid or gaseous. In a similar way during $12^{th} - 13^{th}$ second greater than 15% nodes change to gaseous phase and during $30^{th} - 31^{st}$ second greater than 13% nodes face a phase transition from gaseous to liquid phase. Apart from these apparent critical points, we can observe that beyond 40^{th} second, on an average the ratio of nodes having different phases and sub-phases remains almost the same throughout the experiment. We can conclude that, during the stress, the nodes face frequent phase transitions for a certain period (during $13^{th} - 38^{th}$ second), beyond that the system eventually adjusts with the stress and reaches to a steady-state. We have also computed the average transaction success rate (i.e., transaction which have committed successfully) during our experiment: 42.557% of transactions (with error bar of ± 0.86426) were successful.

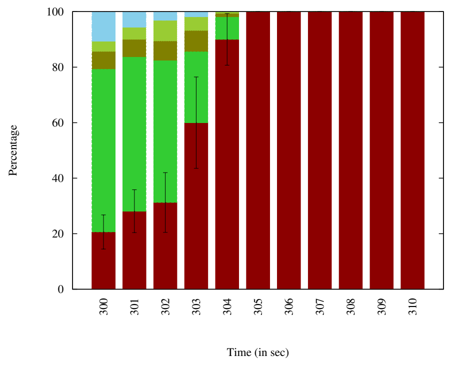
In Figure 10.5b, only error bars for solid phase (P_S) are shown. This part of our experiment shows the weak reversibility of the system, once there is "no stress" in the operating environment. Here, once there is no packet-loss in the physical network, all nodes those are on the branches and are partitioned-away very quickly merge to the core-ring and the recovery completes, i.e., all nodes are in solid phase within 305 seconds.

Heterogeneous Environment: Some Nodes have higher Maximum BW than the others

In this experiment we seek answer to the question: whether heterogeneity (in terms of maximum BW of nodes) in the system has any impact on the phase transitions due to packet-loss in a SON. In order to investigate about this, we start with same experimental setup described before, but this time instead of assigning same maximum BW for each node in the system, the maximum BW of nodes are uniformly distributed in the range $[50, 150]$. The packet-loss model is simulated as described in Section 9.1 for 5 minutes: at a node, p with maximum BW of $BW(p)$ (i.e., p is allowed to send maximum $BW(p)$ messages



(a) Under Packet Loss



(b) After withdrawing packet-loss

Figure 10.6 – Phase Transitions in Beernet (nodes have different maximum BW): red, green (different shades corresponds to 3 liquid sub-phases) and blue (dark, gray and light-gray in B/W) areas correspond to percentages of nodes in solid, liquid and gaseous phases respectively

per second), where $50 \geq BW(p) \leq 150$, for each message with a sequence number of n during a second , if $n < BW(p)$, the message is lost with a probability of $\log_{BW(p)}(n)$, otherwise, the message is lost with probability 1. Every second we take a snapshot of the system, i.e., the percentages of nodes at each phase and sub-phase throughout a run. After 5 minutes we revoke packet-loss, we let the system run with "no stress" until the completion of self-healing (i.e., perfect ring) and every second take the measurements. We have used mean value for 10 independent runs. Figure 10.6a and Figure 10.6b show the states of the system during packet loss followed by zero packet-loss respectively.

Only error bars for solid phase (P_S) are shown in Figure 10.6a. We can observe two differences comparing to the results obtained in Figure 10.1a: (i) the system reaches to the steady-state faster, there is no period during which we can see frequent changes of phase (liquid->gaseous->liquid) of a significant fraction of nodes, as in Figure 10.1a (during $13^{th} - 38^{th}$ second a significant fraction of nodes changes to gaseous phase, followed by another phase transition to liquid); (ii) During the steady-state the percentage of nodes (on an average) in gaseous phase is lower than that in Figure 10.1a, also the ratio of nodes in solid phase has also increased slightly. As expected due to presence of nodes with higher BW (i.e., some nodes experience lower packet-loss rate than the others), the system shows better resilience against the stressful operating environment. However, the successful transaction rate is reduced in this experiment: 35.779% (with error bar of ± 1.3403) of transactions were successful. So, though more nodes are in solid or liquid phase during the stress, that has not improved the available higher-level functionality (e.g., transactional functionality which requires coordination among multiple nodes) of the system, only the connectivity among nodes, i.e., the embedded graph in the system, is in better state during the stress, which implies other lower-level operations (e.g., DHT operations, efficient routing) should be better than the scenario Figure 10.1a, verification of which is left as future work.

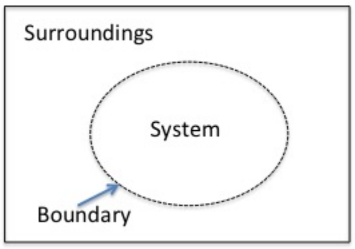


Figure 10.7 – A Thermodynamic System

The trend in Figure 10.6b (only error bars for solid phase are shown) is the same as in Figure 10.5b. As there is no packet-loss in the system, all nodes quickly converge to a perfect ring.

10.4 Analogy with the Thermodynamic System and its Environment

In this section we study the analogy of our system with a thermodynamic system. In thermodynamics both a thermodynamic system and its environment are considered [114], as shown in Figure 10.7. A thermodynamic system is always enclosed by walls that form the boundary of the system. In our context, the distributed system, in particular our representative overlay network is the system and the physical network, e.g., the Internet and other systems running on that physical network consists the environment of the system. We consider the network dynamicity due to packet-loss as the stress element for our system in this section, which we say in analogy with a thermodynamic system, "the heat source" in the environment. In thermodynamics, if the external condition changes, the system responds by changing its state: the temperature, volume, pressure, and chemical composition, and adjust to a new equilibrium [114]. Here, equilibrium implies a state of balance. As we have observed in Section 10.3.2, during packet-loss in the environment, the system eventually reaches to a steady-state, i.e, each node modifies their local states according to their local environments and the overall global state of the system (the ratio of nodes in different phases and sub-phases) continues to be the same throughout the experiment. Thus, we remark that the system reaches to an equilibrium as a response to the change in its environment. In this case, the environment has worked on the system, i.e., due to packet-loss in the environment the internal energy of the system, in other word, the *entropy* of the system has increased. Here, we define *entropy* as the lack of order or predictability in the system.

According to the *First Law of Thermodynamics*, which is the law of conservation of energy, the change in a system's internal energy = the heat absorbed from the environment − the work done on the environment. In other words, using the internal energy the system can work on its environment, e.g., when a gas expands, it works on the external environment. Till now, we have only considered scenarios, where the environment has worked on the system, and due to packet-loss the entropy of the system has increased and reached to an equilibrium (see Section 10.3.2). We seek answer to the questions: what happens if the

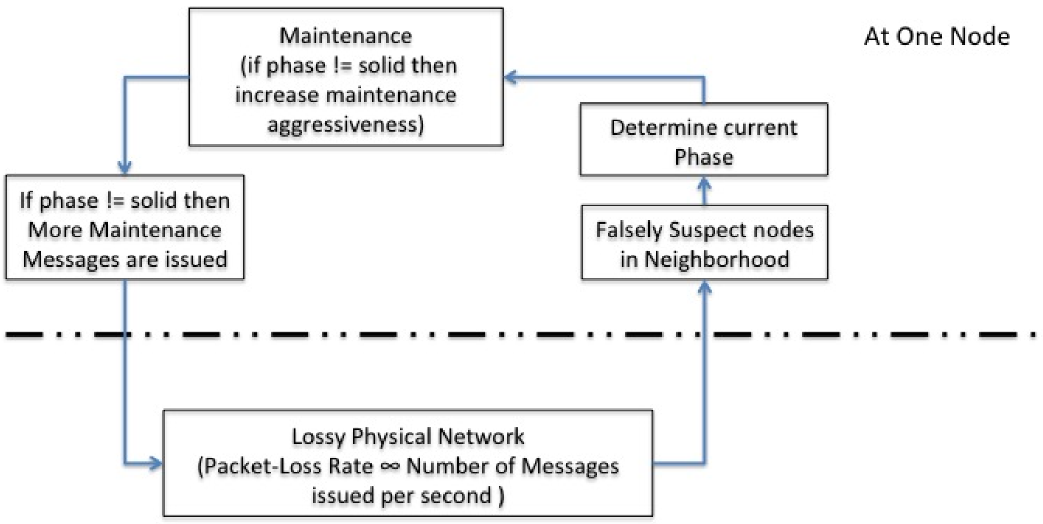

Maintenance
(if phase != solid then
increase maintenance
aggressiveness)
At One Node
Determine current Phase
If phase != solid then More Maintenance Messages are issued
Falsely Suspect nodes in Neighborhood
Lossy Physical Network
(Packet-Loss Rate ∞ Number of Messages issued per second)

Figure 10.8 – Feedback Loop between the system and its lossy environment

system in an attempt to reduce its entropy work on its environment, thus further increases the entropy of its environment? Does the system reach to an equilibrium, while trying to do so? In this section, we conduct experimental studies to answer these questions. For this purpose, we have made the maintenance in our system to be adaptive, i.e., if the current phase of a node is not solid, it gradually increases the aggressiveness of its maintenance. The aggressiveness of the maintenance of the system can be increased by adjusting the system parameters, for example, for all periodic maintenance, the periods of the operations are decreased so that the maintenance becomes more frequent. During packet-loss, false suspicions are triggered, as a result of which, nodes change to liquid phase, also some nodes are partitioned-away, thus face a transition to gaseous phase. As a response to such phase transitions, the nodes make their maintenance more aggressive. Thus, more maintenance messages are issued, increasing the entropy of its environment and more packets are dropped. In other words, in this experiment, due to packet-loss the environment works on the system, increasing the system's entropy, followed by the system working on the environment in an attempt to reduce its entropy. So, we can see a feedback process in effect, where both the system and its environment work on each other, or affect each others' entropy. This feedback loop is schematically depicted in Figure 10.8.

In this experiment, we have used the similar experimental setup as described in Section 10.3.2. For Periodic Stabilization, the period is initialized as 10 seconds. For merger, the period used to dequeue the elements to generate *mlookups* at each node is initialized as $\gamma = 30$ seconds. The number of *mlookups* generated every γ second is initialized as 1. The KB sampling frequency (see Section 3.3.5), $\sigma = 60$ second. Every 5 second, if a node determines its phase to be not solid (P_S), it decreases all these maintenance periods by 1 second, until a period is set as 1. The number of *mlookups* generated every γ second is increased by 1. The duration of a run is 5 minutes and every second we take a snapshot of the system, i.e., the percentages of nodes at each phase and sub-phase throughout a run. We have run different experiments by varying the max BW of the nodes in the system. In an experiment we have used the same max BW for all nodes in the system. The max BW used are 50, 100 and 150 messages per second. For each of these max BW, we have used three different transaction rates: 1, 2 and 3 transactions per second, in order to understand the effect on the highest-level functionality, and also we study how increasing the number of messages triggered by data-level operations impact the overall behavior of the system. A transaction reads one key and updates another one. We present the results of 9 different scenarios in Figure 10.9, Figure 10.10 and Figure 10.11, each result uses mean value for 10 independent runs. Also, we present the percentage of successful transactions (i.e., transactions which are committed) during these 9 experiments in Figure 10.12. We remark that, after revoking packet-loss, the system has exhibited weak reversibility for each of these scenarios, however we have not included the results, as the trend remains the same as that in Section 10.3.2.

As we can see in Figure 10.9, irrespective of the transaction rates, a zigzag pattern in quite explicit in the results. This is because the stress and the maintenance are pulling the system in opposite direction, i.e., the stress is increasing the entropy of the system, while the maintenance of the system is trying to decrease the entropy. We can also see that with time the zigzag pattern diminishes and the system seems to reach a steady-state, i.e., the ratio of nodes in different phases on an average continues to be the same with time. We also observe that for transaction rate of 1 (see Figure 10.9a), the corners of the pattern are sharper than the others. This is because the messages during a second are dominated by the maintenance messages, rather than data-level messages. As a result more maintenance messages are lost, affecting the phase determination of a node, thus triggering more maintenance messages. Due to the same reason, for transaction rate 3 (see Figure 10.9c), the system reaches to the steady-state faster than the others. The lost messages has higher percentage of data-messages, thus the number of maintenance messages which are dropped goes down. Finally, comparing to Figure 10.10 and 10.11, the percentage of nodes in gaseous phase is higher in Figure 10.9. The percentage of successful transactions decreases with the increase in transaction rate, as we can see in Figure 10.12. This is because, as the transaction rate goes higher more data-messages are dropped, resulting in abort of transactions, however the drop in transaction success rate is lower between transaction rate 2 and 3 than for the transaction rate between 1 and 2. Also, the percentage of successful transaction has increased by 10%, compared to the experiment result for the same scenario, except that the maintenance of the system was non-adaptive, as shown in Section 10.3.2.

As we can see in Figure 10.10, increasing max BW has diminished the sharpness of the corners of the zigzag pattern, however, the percentage of successful transaction has also reduced for all transaction rates compared to their counterpart experiments, where the max BW for all nodes were half. In other words, due to increase in max BW, less maintenance messages are dropped, as a result the system reaches to a steady-state faster. Due to this same reason the result in Figure 10.10a, with lower transaction rate than the others (Figure 10.10b and 10.10c) has sharper corners. Also, the percentage of nodes in gaseous phase is lower than those in Figure 10.9. Thus, to summarize we can say that making the max BW of nodes two times has improved the connectivity among nodes, thus the extra BW or lower packet-loss rate has caused better recovery, thus less damage to the embedded graph of the system, however the availability (when tried once) of the highest-level functionality, i.e., transaction success rate has not improved.

The result in Figure 10.11 follows the same trend: making max BW of nodes three times has diminished the sharp corners in the result, less nodes are in gaseous phase. The system stays in an steady-state starting from the 20^{th} second. However the percentage of the successful transaction (see Figure 10.12) has dropped slightly compared to those with max BW of 100. Also, the result with transaction rate of 1, Figure 10.11a, has sharper corners than those in Figure 10.11b and 10.11c.

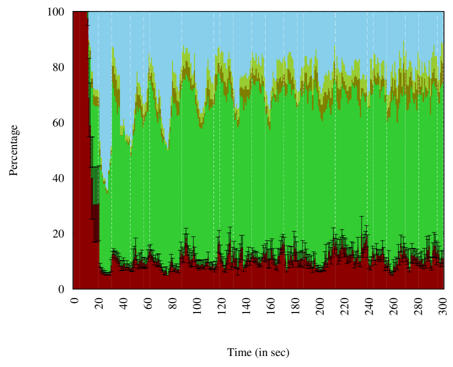
10.5 Related Work

Diligent search has failed to uncover any empirical work on phase transitions in structured peer-to-peer network, however we have found one analytical work [36]. The result of this study, carried out for Chord in [36], shows a critical point in parameter space at which the system with high probability breaks down, i.e., efficient routing becomes impossible. Such phase transitions happen due to high churn and large link delays, resulting in a finite fraction of the connections to be always incorrect/dead. We have come across works on phase transitions in other networked systems during our search. For example, the analytical work in [115] is based on the theory of critical and complex systems, which studies and applies the phase transitions phenomenon for unstructured peer-to-peer networks. In [116], Scholtes et al. present distributed monitoring and adaptation schemes of macroscopic statistical network parameters using power law networks. Such adaptation of critical parameters can be termed as phase transitions. For phase transitions in other topologies of network; for example the small-world model [117, 118, 119, 120, 121, 122], several authors have studied an Ising model. There are several studies [123, 124, 125] on phase transitions in models of Internet traffic. The outcome of these works can be applied in the context of SONs.

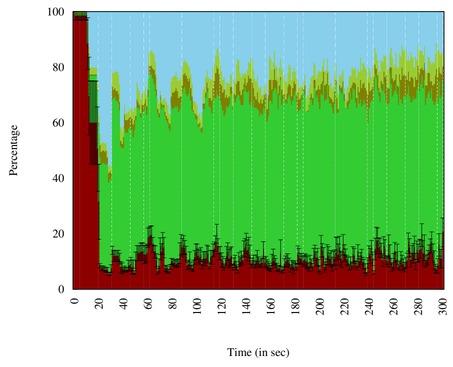
In [47], John Kubiatowicz has presented a novel and insightful characterization of the approach of designing large-scale peer-to-peer systems from the point of view of thermodynamics. As he suggested, properly designed systems can exhibit stable behavior by exploiting multiple components and that these systems have thermodynamic descriptions. In his paper [48], George Fletcher has also emphasized the necessity of gaining insight about designing peer-to-peer systems in analogy to thermodynamic systems. Scholtes et al. in [49] discuss overlay networks management from the perspective of thermodynamics. As discussed in this article, structured approaches to construct overlays (as in the case of our representative system) give rise to states of small entropy, as such approaches make accessible only a small subset of all possible network realizations. However, these overlays require maintenance schemes to prevent gradual loss of structure, and thus increase of entropy. In analogy with thermodynamics, they have referred the overhead induced by the maintenance mechanisms as the input of energy that is used by non-equilibrium biological systems to prevent the increase of entropy due to the second law of thermodynamics. In a similar way, they have considered the unstructured ways to construct overlays corresponding to the states of maximum statistical entropy, as in this case all network realizations are equally likely. They have also discussed a middle ground, i.e., construct overlays with intermediate levels of entropy. Such overlays are neither completely random nor deterministic, rather introduce a statistical structure, e.g., a particular type of degree sequence, a certain clustering coefficient, correlations between data location and network structures etc. The analytical work in [126], identify the scale-free variables in analogy with thermodynamics and statistical mechanics. In physics, the problem of reasoning about large systems with many components has already been dealt with. In analogy, in this paper the authors try to identify the intensive (i.e., variables that become independent of system size) and extensive (i.e., variables that become proportional to the size of the system) variables in the context of structured overlay network, in particular Chord [11]. They have also discussed the idea of identifying phases in the system, where in each phase all intensive variables vary smoothly, and where the characteristics of the system remain the same. All these works have emphasized the need of constructing and analyzing large-scale distributed systems by borrowing the knowledge and techniques from the physics, in particular thermodynamics. Our empirical studies are an attempt to extend this analogy further, and we want to continue further towards this direction in our future work.

10.6 Discussion

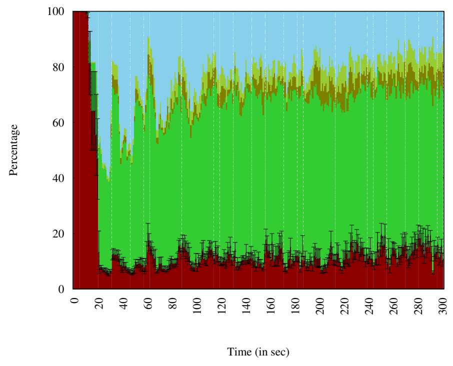
In this chapter we have proposed an efficient approach to approximate the available functionality, i.e., the reversibility function for our representative system. As per our approach, based on the phase of each node, the operations currently available of the system can be approximated without any extra distributed operation. Next, we have applied the definition of phase to identify different phases, sub-phases in Beernet. We have experimentally demonstrated reversible phase transitions in Beernet due to increase/decrease of stress intensity. We have used churn and packet-loss as the stress elements in this chapter. Such phase transitions are a consequence of having a reversible system. Also, we have analyzed the critical points observed in our experiments. Finally, we have presented an experimental study to compare the feedback analogy with a thermodynamic system and its environment, using network dynamicity due to packet-loss as the stress element. As we have observed in these experiments, the global state of the system exhibits zigzag pattern as the maintenance and the stress pull the system in opposite directions, the stress tends to increase the system entropy, whereas the maintenance aggressively attempts to decrease it. We have also noticed that increasing max BW of the nodes and number of transactions in the same rate does not improve the success rate of transactions, however improves the connectivity, thus, better and faster recovery of the embedded graph of the system.



(a) Max BW: 50 messages per second;
Transaction Rate: 1 per second

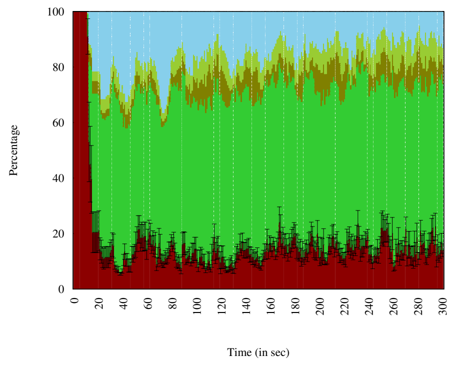


(b) Max BW: 50 messages per second;
Transaction Rate: 2 per second

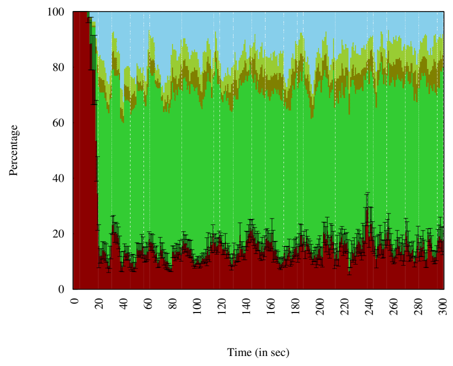


(c) Max BW: 50 messages per second;
Transaction Rate: 3 per second

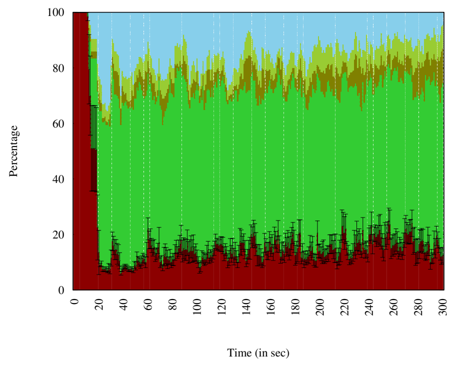
Figure 10.9 – Phase Transitions in Beernet (each node has maximum BW of 50 messages per second): red, green (different shades corresponds to 3 liquid sub-phases) and blue (dark, gray and light-gray in B/W) areas correspond to percentages of nodes in solid, liquid and gaseous phases respectively



(a) Max BW: 100 messages per second;
Transaction Rate: 1 per second

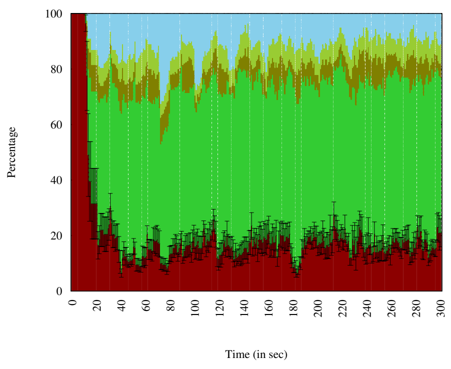


(b) Max BW: 100 messages per second;
Transaction Rate: 2 per second

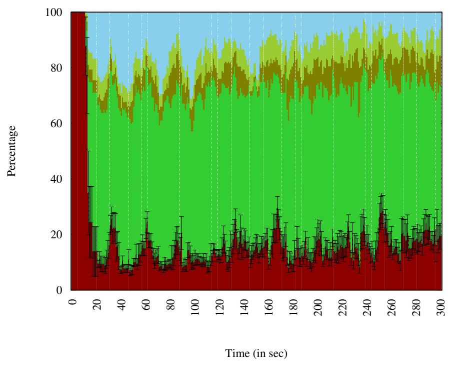


(c) Max BW: 100 messages per second;
Transaction Rate: 3 per second

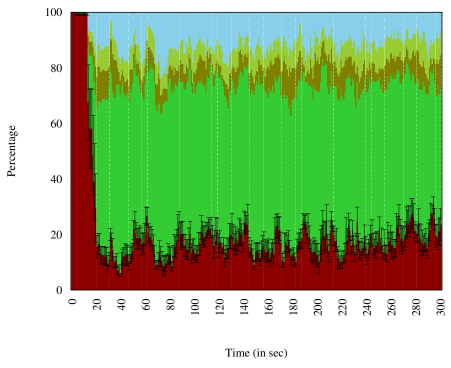
Figure 10.10 – Phase Transitions in Beernet (each node has maximum BW of 100 messages per second): red, green (different shades corresponds to 3 liquid sub-phases) and blue (dark, gray and light-gray in B/W) areas correspond to percentages of nodes in solid, liquid and gaseous phases respectively



(a) Max BW: 150 messages per second;
Transaction Rate: 1 per second



(b) Max BW: 150 messages per second;
Transaction Rate: 2 per second



(c) Max BW: 150 messages per second;
Transaction Rate: 3 per second

Figure 10.11 – Phase Transitions in Beernet (each node has maximum BW of 150 messages per second): red, green (different shades corresponds to 3 liquid sub-phases) and blue (dark, gray and light-gray in B/W) areas correspond to percentages of nodes in solid, liquid and gaseous phases respectively

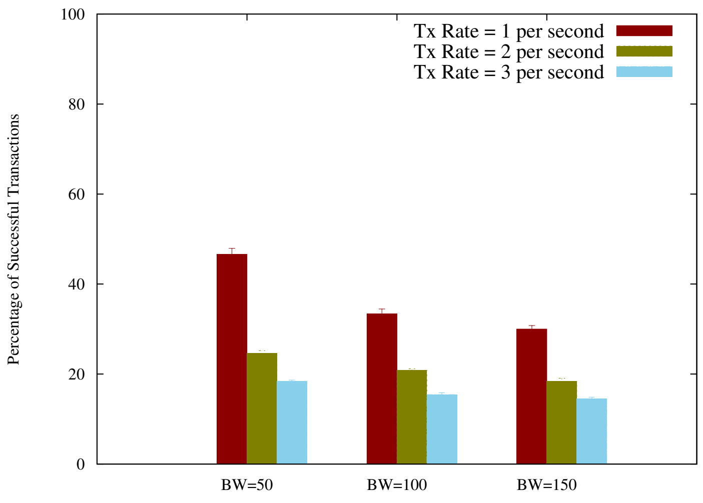


Figure 10.12 – Percentage of Successful transactions for various max BW and transaction rate

Chapter 11

Phase API and Application

We have introduced the concept of *Phase* (See Section 2.2) and explained (see Chapter 10) using our representative system, Beernet, how the available operations of the system can be efficiently approximated based on the phase of an individual node. In this chapter, we focus on designing applications on reversible systems so that taking advantage of the reversibility of the underlying system the application can manage its behavior while experiencing stress in its operating environment. We analyze the design decisions of the application which will allow it to have enhanced self-adaptation, self-optimization and achieve reversibility at the application level as well as the SON level.

In this chapter, we explain how a node determines its phase without any global synchronization. The current phase of a node should capture enough information about the stress in that node's local environment and also should convey sufficient information about the functionalities currently available. Next, we present an API so that the application can access the current phase of a node and be notified when a phase transition occurs. We describe phase-aware application design. We analyze how the application can be designed to be *Reversible* itself, based on the concept of phase. We draw general conclusions about the usefulness of the "phase" concept towards designing predictable and reversible application services. We describe the phase-aware design of a nontrivial application. We show the enhancements obtained due to the phase-aware design decisions.

11.1 Computing Phase at Individual Node

The *Phase* is a local property of each node. Each node determines its own phase independently and at a given instant different nodes in the system can be in different phases. The current phase of a node should reflect the current stress the node is experiencing in its local environment. Also, the phase determination at each node should work with low extra computation and resource consumption, so that it can be useful even in highly stressful scenarios. Under all these constraints how can a node determine its current phase with high confidence, without any global synchronization? In order to determine the current phase at each node by satisfying these constraints, *self-awareness* is crucial for each node. In this section we describe how we have extended each node in our representative system to be self-aware and how a *self-aware* node determines its phase following the semantics described in Section 10.2.

Self-Awareness Knowledge about only its own internal state is not enough for a node to determine its phase with high confidence: because the node will have no knowledge about its immediate vicinity. A node also needs information about how it is being perceived by the other parts of the system. The combined knowledge from both of these sources can provide meaningful context for behavioral decisions and prediction, thus enable a node with optimal phase determination. These two types of knowledge can be termed as *private* and *public self-awareness* respectively following the definition of *distributed self-awareness* [127] by [128]. A *self-aware* node can determine its phase at any instant with high confidence without any global synchronization.

Internal State of A Node: We say the internal state of a node, with id p, of a SON is $State_p = \{p, N(p)\}$; where $N(p)$ is the neighborhood (see Section 3.2) of p. For our representative system, $N(p) = \{successor(p), predecessor(p), RT_p\}$, here RT_p is the routing table of p. $N(p)$ is time-dependent due to the dynamics in the overlay. So, each node corrects its internal state, if required, as part of its maintenance strategy.

Achieving Public Self-Awareness: We have extended each node to extract information about how it is being perceived by the other nodes in the system. A node achieves this form of *public self-awareness* as part of its maintenance and also from the messages that it receives or that are routed through it. The extracted knowledge is stored as: $K_{public} = \{Br, Br_{root}, Br_{tail}, Br_{size}, Br_{depth}\}$, where Br, Br_{root}, Br_{tail} are flags denoting existence of a branch and whether the current node is the root or tail of a branch respectively. Br_{size} and Br_{depth} denote the size of the branch and the depth of the current node on the branch. In Beernet each node maintains a list of nodes, called *predList*, that consider the current node as its successor, in order to do routing on the branches. The *predList* is used to determine the branch size and a node's depth on the branch. The messages used to achieve public self-awareness can be both maintenance and protocol messages due to application-driven actions. For example, consider a node with id u and predecessor s (see Figure 3.2). So, u is responsible for the keys $\{t, u\}$. Now, if u receives a message (and the message is destined to u based on the flag as described in Section 3.2) concerning a data item with key q, u can easily imply that (some) other nodes of the system perceive it to be responsible of key q. Thus, u can gather the knowledge about the existence of a branch in its local environment and sets the corresponding flag. Similarly, public self-awareness is achieved as part of maintenance also, particularly via proactive maintenance. For example, as part of Periodic Stabilization (PS), each node exchanges periodic messages with its successor to maintain its internal state. Consider a node with virtual id y and predecessor x (see Figure 3.2). If y receives a PS message from w, which implies w perceives y as its successor and is not aware of or suspecting node x. Thus, y can learn the knowledge of x being on a branch and be the root of a branch itself, thus it sets the corresponding flags. Similarly, x can also deduce from the absence of periodic maintenance messages from its predecessor about it being pushed as a tail on a branch, thus it sets its flags.

Periodic Phase Computation As mentioned in Section 2.2, phase at each node is a function of time. So phase determination at a node is periodic. Every T_{phase} time unit, a node computes its current phase. T_{phase} is a configurable parameter. If this time window is too long, several phase transitions, that the node has experienced in between, might be missed to be traced, especially when there is stress in the operating environment. As a result, the application layer also will not be notified about those, which might affect the performance of the application that invokes phase-driven actions. A better approach is to recompute phase whenever there is a change in a node's internal state or public self-awareness, implementation of which is left as future work.

We can express the phase determination at a node, p, during a round R, using a function F_{det}. The phase of p at round R is, $Phase_p(R) = F_{det}(State_p(R), K_{public}(R), History_{phase}(p, S))$. Here, $State_p(R)$ is the internal state of p and $K_{public}(R)$ is the knowledge p has gathered during T_{phase} time unit. $History_{phase}(p, S)$ is a vector which stores the phases of p during the last S rounds. This is required particularly to optimize the decision about the routing table in our context (explained shortly). The function, F_{det}, can be a simple algorithm, such as if-else logic on the fields of the parameters, or a stronger machine learning algorithm. In our implementation, we have used light-weight if-else logic on the flags and values of the fields of three parameters to determine phase at each node. The goal is to demonstrate that using simple logic at each node, without introducing extra resource consumption, the usability and behavior of a distributed application can be significantly improved.

As we mentioned in Section 3.2, each node perceives the identifier space to be partitioned into $\log_k(N)$ partitions, where each partition is k times bigger than the previous one. The routing table of a node contains $\log_k(N)$ fingers to some nodes from each partition. In co-relation with such k-ary routing table, we have assigned weights to the fingers, depending on the partition it belongs to, to compute the invalid/missing fingers. The longest finger to the biggest partition has a maximum weight of $1/k$ and for the next k time smaller partition, the finger weight is also reduced proportionally, i.e., $1/k^2$ and so on. The reason behind such weighted scheme to compute invalid fingers is that if the longest finger is missing or suspected in a node's routing table then it affects the routing guarantees more than an invalid finger to the smallest partition, which is in its immediate neighborhood. Now, consider a scenario: in node p's routing table, one node x is responsible for more than one contiguous partition. As a result p will find missing fingers to some partitions, thus might consider its routing table to be in not-so-good-state (e.g., $> 50\%$ missing fingers) and compute its phase during that round accordingly. However, it might be the case that the routing table at p is in perfect state as per the global view and the reason behind the missing fingers to some partitions are something different than the effect of stress in the operating environment, e.g., there are only few nodes in the system or skewed mapping of the nodes on the identifier space. In order to take these scenarios into account p must revise its decision about its routing table in subsequent rounds: if during contiguous S rounds, the routing table is in the same "not-so-good" state, then p can safely assume that this is the optimal state of the routing table (p can also proactively verify this, thus optimize its decision, by doing lookups using the ids of the invalid/missing fingers; however also incurring resource consumption) as per the global view and re-evaluate its phase decision. Here, S

is a configurable parameter: if S is too low then, a node might misjudge about a scenario, whereas for very high values of S the node will take more time to re-evaluate its phase, also it will consume more memory to store the history.

11.2 API: Exposing Phase to Applications

We provide qualitative indication of what system operations are currently available, i.e., $Op_{avail}(t)$, by exposing the phase of a node. The phase information is given to the application at each node, using an API. Our API supports push and pull methods: using pull method the application queries the underlying node about its current phase, whereas the push method allows the application to be notified when the underlying node changes phase. A node can be in one of these phases: P_S, P_{L1}, P_{L2}, P_{L3}, P_G, the semantics of which are described in Section 10.2.

- ***getPhase(?P_{cur})*** Binds P_{cur} to the current phase of the peer.
- ***setPhaseNotify(f)*** Sets a user-defined function, ***f(?P_{new})*** to be executed as a phase transition happens. Upon a phase transition, P_{new} is bound to the next phase of the peer and ***f*** is executed. Executions of ***f*** are serialized in the same thread over a stream of successor phases.

With this API, the application can be made reversible, i.e., the application can monitor the qualitative network behavior and change its own behavior accordingly.

11.3 Application

The upper layer (in particular the application running on top of a SON) can use the information of phase and phase transition of a node towards making the system reversible and predictable. We discuss some applications of the concept of phase in these two directions. Next, based on the phase of the underlying node, we analyze the design of a use-case application, a collaborative graphic editor, with enhanced self-adaptation and self-optimization properties. Furthermore, we analyze how the application itself can achieve reversibility in the application-level semantics. Using the phase of the underlying node, the application provides indication to the user regarding its behavior (current and to be expected). Thus, the application has improved behavior with respect to the user, i.e., the user can understand and decide what to do in a high-stress environment. *Reversibility* gives improved functionality (internal to the application) and *Phase-Awareness* gives improved usability (external user).

- **Reversible System:** The application running on top of a SON can itself achieve reversibility based on the phase and phase transition of the underlying node. For example, if the current phase of the node is *liquid-3*, then it is highly probable that the node might face a phase transition to gas in near future, thus the application can trigger additional replication of the data stored on that node, either to a persistent storage or at the root of the branch based on the replication strategy. Thus, the application can avoid possible loss of data and able to achieve reversibility by itself. Further, the system is able to conduct efficient self-healing using the phase information of a node. As the phase of a node is a reflection of current stress in its local environment, the node can adapt its maintenance as per its current phase. For example, a node in *liquid-2* needs to be more proactive to do self-healing than a node which is in *solid* phase. Thus, when a node faces a phase transition from *liquid-1* to *liquid-2*, it can trigger aggressive proactive local maintenance by reducing the period of its periodic stabilization maintenance strategy. Similarly when a node is in *solid* phase, it can turn-off its proactive maintenance to avoid unnecessary bandwidth consumption. Thus, using the knowledge of phase and phase transition of each node, the system can have improved self-* properties, and do efficient maintenance to achieve reversibility.

- **Predictable System:** Based on the current phase of a peer, the application running on that node can inform the user regarding the current available guaranteed functionalities. Thus, the user is able to get feedback and prediction about the dynamic behavior of the system. Also, at the same time, the application can adapt its behavior based on the available functionalities from the underlying SON. All these can lead to designing a predictable system .

We illustrate the usefulness of the phase concept using a real application scenario. Consider a *Distributed Version Control System* running on top of SON, which is notified that the underlying node changes its phase. The application notifies the user via an indicator, B_{conn}, that changes its color to indicate the phase of the node. B_{conn} can be green or yellow or red, denoting respectively *solid/liquid/gaseous* phase of the node. Suppose the user using this system on a network having intermittent connectivity (e.g., Wi-Fi on a fast train). As long as B_{conn} is green, the user can continue her work without being concerned. However, as B_{conn} changes to yellow, the user can initiate a $pull$ to retrieve the most recent version and $push$ her own changes. These allow the user to work productively offline on the up-to-date version and prevent any potential data-loss. Thus, the application itself can achieve reversibility as the connectivity of the underlying network is restored, and the user is able to predict the behavior of the system. For example, the application (e.g., real-time collaborative editing) can adapt the philosophy of exponential back-off as TCP congestion algorithm, which is now brought up to the user level in terms of phase. Further, the underlying node can adapt its maintenance to the current phase leading to efficient self-management. For example, when a node is in P_S phase, an efficient strategy like correction-on-* is sufficient. As the node faces a transition to P_{L1}, it can turn on a more resilient strategy, like PS.

11.3.1 Phase-Aware Application Design: A Use Case

We have selected a nontrivial use case to empirically demonstrate phase-aware application design for high-stress operating environments. In this section we describe our use case application, *DeTransDraw = Decentralized Transactional Drawing (DTD)*, a collaborative drawing tool. The first version of DTD is described in [13]. In this thesis, we extend DTD to integrate phase-aware design decisions, as a result of which the application has achieved improved self-management properties, better usability and reversibility. We describe and analyze the phase-aware design decisions integrated with DTD.

DTD is a decentralized real-time collaborative vector-based drawing application. It has a graphical editor with a shared drawing area. There is coherent collaboration among the users, where each user is graphically notified about the activities of others. DTD is the successor of *TransDraw* [129], which has a client-server architecture. DTD is the decentralized version of TransDraw that runs on a peer-to-peer network. In TransDraw, as the server holds the whole state of the application, it is the single point of failure as well as the source of congestion and scalability issues, whereas for DTD the application state is spread across the network being symmetrically replicated (our underlying SON uses *Symmetric* Replication [70]). Thus, DTD provides load balancing, scalability, and fault tolerance.

DTD has a shared drawing area, where all users can access all figures of the drawing. To manage conflict resolution among users and reduce the problems of network latency, transactions are used. The goal is to allow a user to modify the figures immediately, without waiting for the confirmation of a distributed operation. The conflicts (if they arise), will be solved by the transaction manager later. However, due to such optimistic approach (from a user's perspective) a user may lose her modifications if there are concurrent modifications of the same figure by another user, or the user is experiencing intermittent connectivity (i.e., the application has stress in its operating environment).

DTD is an asynchronous and consistent (with conflict resolution) collaborative application, which implies if two users concurrently modify the same object, then at the end of their works only one will be able to successfully commit her changes, while the other will lose her modifications. The probability of encountering such scenarios is much larger in coherent collaboration. So, we need a pessimistic transactional approach with eager locking, so that the user can get notification about any concurrent update earlier, instead of losing all her modifications when trying to commit, after finishing all the works. A Paxos consensus [112] based commit protocol [94] with optimistic locking, where the client performs some modifications, then requests corresponding locks to commit the changes, hinders the functionality of such applications. So we use an adapted version [13] of Paxos to support eager locking (referred as *Eager Paxos*), with a notification mechanism so that other users (i.e., registered readers) get notifications of updates or locking of every shared item.

Figure 11.1 shows Eager Paxos transactional protocol as used in DTD. An Eager Paxos transaction is done in two steps: i) get the lock and ii) commit. When a user starts modifying one or more objects, the client initiates a transaction by contacting a transaction manager (TM). The TM is different for every transaction. Before the TM requests for lock, it registers a set of replicated transaction managers (rTMs) for fault tolerance. If the TM crashes, the rTMs carry on the transaction. To get the lock, the TM contacts all transaction participants (TPs). A TP is a node which stores a replica of an object involved in the transaction. Once the decision (lock granted or rejected) has been made, the client is informed, to prevent users from working on already-locked objects. If the locks are granted, the new values of corresponding items are committed, at the end of user's modifications. DTD merges optimistic and pessimistic approaches. The application has an optimistic approach

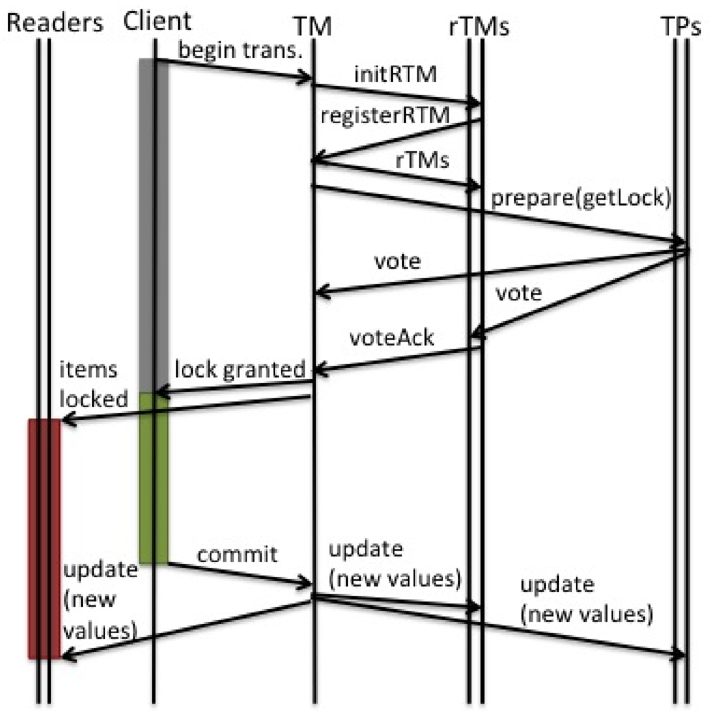


Figure 11.1 – DeTransDraw Transactional Protocol (Eager Paxos with notifications to the readers)

because the user does not need to wait for the lock to start working and it is pessimistic in its transactional protocol, where a transaction first tries to get the lock, then commits the changes.

Figure 11.2 shows the graphical user interface (GUI) of DTD being run by a client. The GUI has four parts: the title bar, the canvas (which is the shared drawing area), the toolbar and the status bar. The title bar shows the virtual identifier of the underlying node. The toolbar consists of five buttons. The purpose of the two colored buttons, from top to bottom, is to set the color of the object and its border respectively. As the labels suggest, the buttons: “Draw rectangle” and “Draw oval” allow the user to draw rectangles and ovals. These are the only figures currently supported by DTD. Using the “Select” button the user can select an object. Multiple objects can be selected by holding the *Shift* key while selecting the objects. When the user selects one or more objects, the corresponding object(s) have eight red dots surrounding it on the GUI and the client requests for corresponding locks. If the locks are granted the red dots surrounding the objects become black; otherwise the red dots disappear, implying dis-allowance of selection of those objects and also, the modification(s) to these objects are aborted, reverting to their original states. As we can see in Figure 11.2, the top-right oval is successfully selected by the user. As the user unselects the selected objects, the new state of those objects are committed (thus, other users can observe them), locks are released and the black dots of selection disappear. The status bar notifies the user about two things. *A phase-aware indicator* (color of the status bar) conveys to the user the

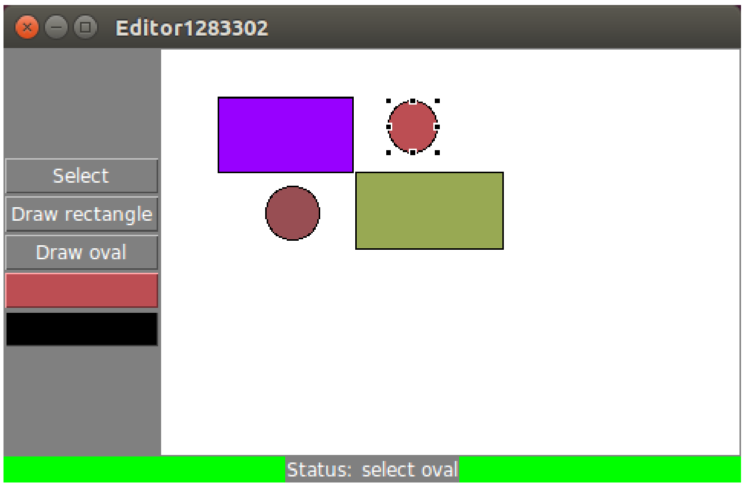


Figure 11.2 – DeTransDraw GUI: Phase-driven behavioral indication via the color of the status bar at the bottom, green/yellow/red (gray/light-gray/dark in B/W) denoting respectively *solid/liquid/gaseous* phase of the underlying node

behavioral prediction of the application (for details see Section 11.3.1). The second part of the status bar notifies the user about her current action. The status bar in Figure 11.2 says that the user is in selection mode and has selected an oval object.

Phase-Aware Design Aspects

In this section we describe how we extend DTD to integrate several phase-aware design decisions. The goal is to enable the application to manage its behavior in stressful operating environments and achieve self adaptation and self optimization. Showing the phase of the underlying node allows the application to become behaviorally predictable to the users.

Phase-aware Behavioral Indication: We have integrated *color-based phase-aware* behavioral indicator in the status bar of our application. The indicator can be green or yellow or red, denoting respectively *solid/liquid/gaseous* phase (see Section 10.2) of the underlying node. From the user's perspective, if this indicator is green, she can continue her work without being concerned. If the indicator turns to yellow, the user can expect some functionality disruptions. The red color of this indicator tells the user about her offline state. The indicator in Figure 11.2 is green, thus assuring the user with full functionality of the application. In this way, the application becomes behaviorally predictable to the user. Such predictability indications allow the users to manage their behaviors and use the application productively, especially when there is stress in the application's operating environment.

Phase-aware Self Adaptation: We extend DTD to react to the phase transitions of the underlying node, thus adapting its behavior with the changes in operating environment. If the application is notified about a transition from *solid* to *liquid*, the application deduces that the underlying system is not stable anymore and reacts. It commits all the pending modifications and releases the locks. For example, when the user is doing modifications on one or more selected objects and a transition from *solid* to *liquid* happens, the application commits the changes till that point and releases the locks of corresponding objects, instead of waiting for the user's action to initiate the commit. The goal is to avoid losing all the user's actions at the end, due to intermittent connectivity. However, if the application immediately takes such reactive actions, it might degrade the environment by making the physical links more congested. Also, the user experience might also be affected, e.g., a transient slow-down of the underlying physical link changing the phase for a brief period; for such scenarios reacting immediately will deter the user's work-flow. To avoid these, the application waits for a certain time, T_{React} (a configurable parameter) before reacting. Suppose, the application is notified about the phase transition at T^{th} time unit, it waits till $T + T_{React}$ time unit, if the current phase is still not a stable one, it initiates the reactive actions. Such adaptive actions will enhance user's experience of the application in high-stress operating environment.

Phase-aware Self Optimization: We extend DTD to proactively optimize its state based on the recent phase transition history. As there is coherent collaboration among the users, keeping the canvas consistent is important. Suppose, there are three users, A, B, and C, using DTD. B is having intermittent connectivity and misses some updates of other users' activities, making B's canvas inconsistent. In a similar way, A and/or C might miss the notifications about B's activities. To avoid such inconsistencies, the application needs to proactively sync a user's canvas with its global state. However, unnecessary synchronization attempts will only result in resource consumption. The application can initiate such optimization efforts based on the recent phase transitions of the underlying node. If the application is notified about a transition to *gaseous* phase or m times *solid* $\leftrightarrow$ *liquid* phase transitions during a period of T_{opt}, the application initiates a read transaction to sync its canvas with the global state. Here, T_{opt} and m are configurable parameters. We have set $m > 1$ to avoid unnecessary optimization attempts (thus, resource consumption) due to transient transitions to *liquid* phase, because for such scenarios the probability that any update is lost is very low. Thus, based on the phase transitions of the underlying node, the application proactively achieves self optimization.

Evaluation

In this section we evaluate DTD in stressful operating environments and analyze the enhancements obtained due to the phase-aware extensions. We discuss the reversibility of the application against such environments. Visual demonstrations of our experiments can be found in [130].

For our experiments, an overlay of 10 nodes is used and a DTD instance runs on each node. During the steady state of the system, we invoke different stresses, e.g., churn, network congestion, and partitioning. We observe the behavior and available functionalities of the application during the stress and after the stress is revoked. For the purpose of making the figures readable, we present screen shots of only four representative DTD instances.

We simulate churn by terminating one or more DTD instances (i.e., injecting failures at the underlying nodes) and joining new instances. We keep the average number of instances the same. Figure 11.3 shows the behavior of our DTD instances when one node fails and is replaced by another node. The indicators on the status bars (of corresponding successors and predecessors) correctly capture these events by showing a transient transition to yellow color (i.e., *liquid* phase). However, due to the reactive maintenance strategy, the concerned nodes correct their states immediately and change to *solid* phase again after the completion of self healing.

In order to assess the behavior of the application when there is congestion in the physical network, we invoke slow down of node communication. We simulate such scenarios by adding delay (30 seconds in this work) on each outgoing message of a slow node. However, as the failure detection at each node of our SON is adaptive (see Section 5.1) (i.e., it adapts with *Round Trip Time (RTT)*), after initial false suspicions, all nodes adapt with the new RTT of a slow node and the system becomes stable again (i.e., all nodes in *solid* phase). To make the operating environment more stressful for the system and simulate intermittent connectivity, we have triggered periodic slow down of node(s). In our experiments, every minute a slow node slows down by 30 seconds for a period of 20 seconds, i.e., during this 20 seconds period each outgoing message of the node suffers an added delay of 30 seconds, followed by a one minute period, during which there are no added delays on the outgoing messages. We have invoked 5 rounds of such intermittent connectivity for 2 slow nodes. Figure 11.4 shows the 3 snapshots of our experiment, where user B and C have intermittent connectivity. As we can see, D is in *liquid* phase as it is suspecting its successor C, and C is in gaseous phase transiently, because due to its intermittent connectivity, C is also falsely suspecting its predecessor, D and successor, A. However, B is in *solid* phase in Figure 11.4b, the reason is: B is in a period during which its connectivity has been restored (i.e., no added delay on its outgoing messages). Also, due to facing congestion, both B and C have missed some activities of A and D (see Figure 11.4b). However, the application eventually recovers all its functionalities at all users, once the system is stable again, as observed in Figure 11.4c.

Next we experiment with network partitioning. We simulate scenarios where node(s) are partitioned away. During the partition, the DTD instance(s) on one partition do not receive any updates from the other(s). As the partition ceases, the application syncs with the global state as part of its self optimization and achieves canvas consistency across users. However, in our current implementation, the users in one partition lose all their modifications done during the partition after the partition is repaired. The reason is our simple design decision: we only retain the highest version of the canvas. However, this could be easily improved, by allowing the application to merge the diverging versions accounting to application-level semantics. Figure 11.5 shows three snapshots of our experiment where a node is partitioned away. As we can see in Figure 11.5b, during the partition, user A, C and D are in one partition, whereas B is in another partition. With time the canvas across the partitions continues to diverge, i.e., the users in one partition are not aware of the activities of the users in the other partition. After the partition ceases, all DTD instances synchronize with the global state (which is the highest version of the canvas), thus eventually restoring canvas consistency (as in Figure 11.5c, though B has lost all her modifications done during the partition) and other functionalities.

Reversibility of DeTransDraw: To analyze the reversibility of DTD, first we need to identify its overall functionalities, i.e., Op_{total}. For DTD, the set of functionalities, $Op_{total} = \{CC, ADD, DEL, SEL, MULTSEL, UPD\}$.

- CC corresponds to consistent canvas across users;
- ADD and DEL correspond to the functionalities of adding and deleting a figure respectively;
- SEL and $MULTSEL$ allow a user to select one and multiple figure(s) on the canvas respectively;
- UPD is the functionality to update one or more selected figure(s).

During our experiments, we have assessed reversibility of DTD. While experiencing stress, some functionalities may become transiently unavailable, e.g., due to lost updates (as a result of congestion or partitioning in physical network) the canvas consistency gets broken; if a node crashes while holding lock of one or more figure(s) (as a result of selecting those), other nodes are temporarily unable to select and update those particular figure(s), until the lock expires. However, we have established the weak reversibility of our application, i.e., when the stress fades away, the application eventually regains all its functionality.

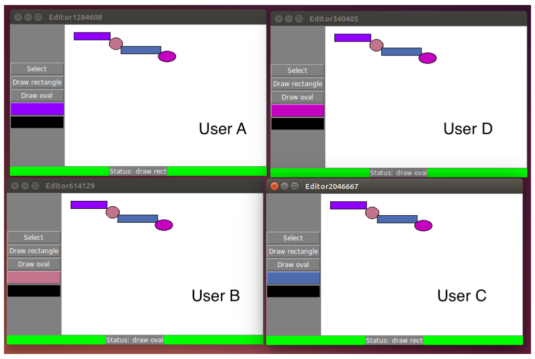
11.4 Related Work

We briefly summarize the relevant work on self awareness in distributed systems. Next, we briefly mention some related works about self adaptation and self optimization in distributed applications. A paradigm shift in system design is explored in [131], [132], from procedural design methodology, where the behavior of the system is pre-programmed, towards self-aware system design, where the system adapts to its context during run-time. In their follow-up work [133], a self-aware computing framework is proposed, which automatically and dynamically schedules actions to satisfy the application's goals. Another approach to build self-managing systems is proposed in [134], [135], based on interacting feedback loops, so that systems can adapt to a wide range of operating conditions and maintain useful functionality. The work in [136] explores the design of a self-aware application to control the behavior of distributed smart cameras. Based on a utility function, the system decides at run-time how to exchange tracking responsibilities among cameras. A much simpler application of self awareness can be found in [137], in which cognitive radio devices monitor and control their capabilities and also communicate others to achieve efficient communication by negotiating changes in parameters.

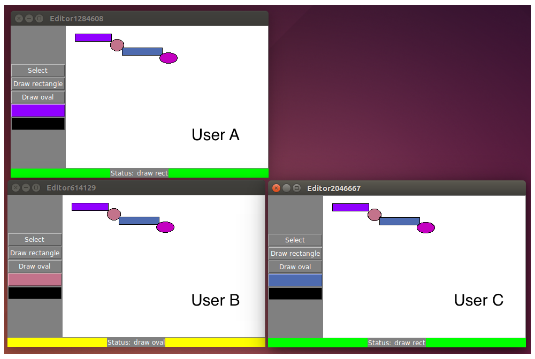
Diligent search has failed to uncover any work on phase-aware application design. However, we have found one analytical work [36] on phase transitions in SONs which shows, using Chord, a critical point in the parameter space at which the system with high probability breaks down, i.e., efficient routing becomes impossible. Several works have been found on designing self-adaptive and self-optimized software systems dealing with distributed applications. The survey in [138] proposes a taxonomy of concerned aspects of adaptation and also presents a landscape of research to identify the research gaps and corresponding challenges. Ledoux et al. in [139] propose an approach, following the principle of Separation of Concerns, to systematically develop self-adaptive component-based applications. In their approach, the adaptation logic is developed separately from the rest and the adaptation policies are interpreted at run-time based on the changes in the environment, detected by a context-awareness service. The work in [140] develops a mechanism to optimize the resource consumption between nodes by transferring services to other nodes. In our work, the context-awareness is provided to the application by the underlying system, in the form of phases, based on which the application can trigger its adaptation and optimization policies, thus can be seen as complementary to these works.

11.5 Discussion

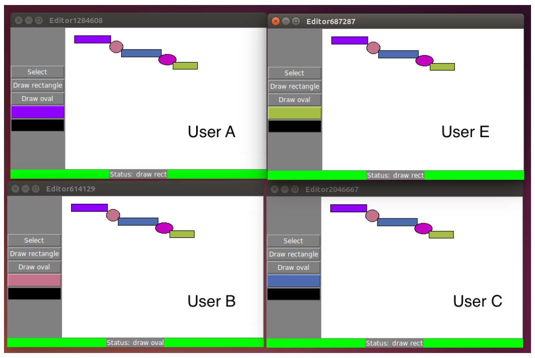
In order to take full advantage of edge computing, large-scale applications must be designed considering the high stress inherent in such operating environments. In this chapter, we have exhibited an approach for designing applications for such environments. We have built our application on top of a reversible and phase-aware structured overlay network, hosting a transactional DHT. We have applied and expanded the concept of reversibility in the context of application design. We extend each node of the underlying system to be self-aware and demonstrate how a self-aware node can determine its phase without any global synchronization. We have applied the concept of Phase to approximate the available operations in our system, and have demonstrated how this information can be made available to the application layer by exposing the phase of the underlying node. We have described phase-aware design of a use-case application. Using this use-case application, we exhibit, based on the phase and phase transitions of the underlying node, how an application can achieve reversibility in the application-level semantics. Furthermore, the application can have improved behavior from the viewpoint of the user, i.e.: have self adaptation, self optimization, and, be behaviorally predictable to the user in high-stress operating environments.



(a) 4 instances of DeTransDraw

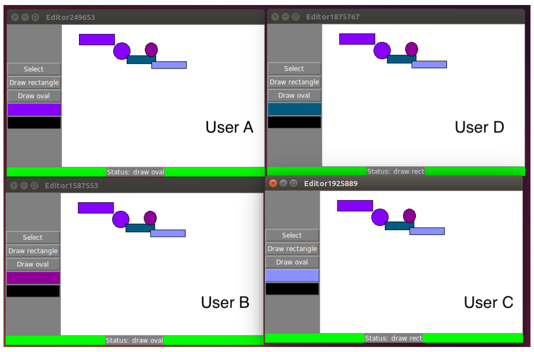


(b) User D has crashed

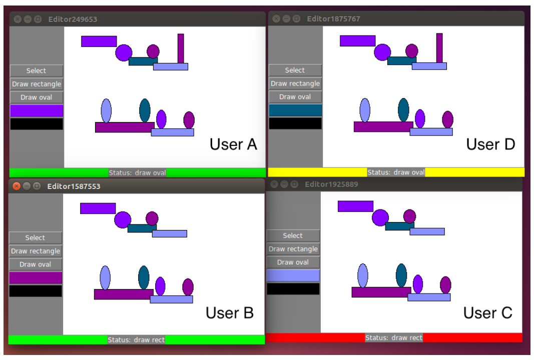


(c) User E has joined

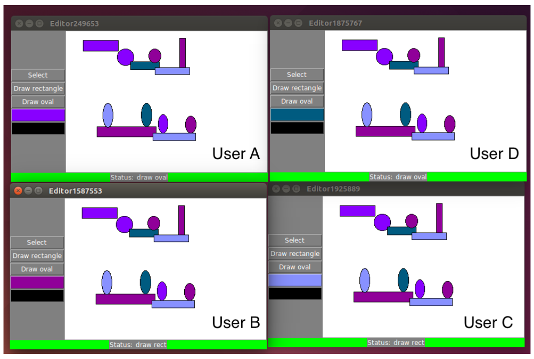
Figure 11.3 – DeTransDraw facing Churn: Suppose User A, B, C and D (as annotated). User D has crashed and replaced by E. As a result, B faces a transient transition to *liquid* phase while doing self-healing, as D was its successor on the overlay (Figure 11.3b). The system is stable again with full functionality (Figure 11.3c).



(a) 4 instances of DeTransDraw

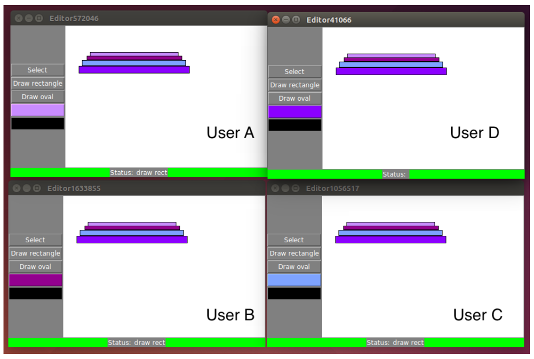


(b) Intermittent connectivity of B and C

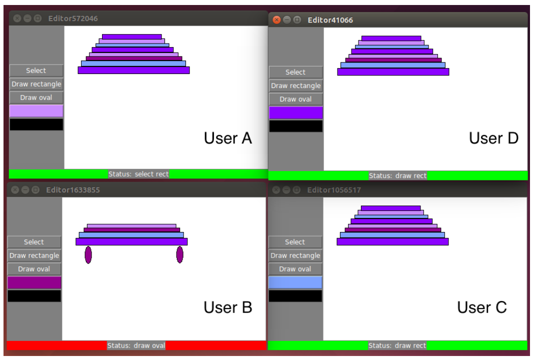


(c) Eventually stable with full functionality

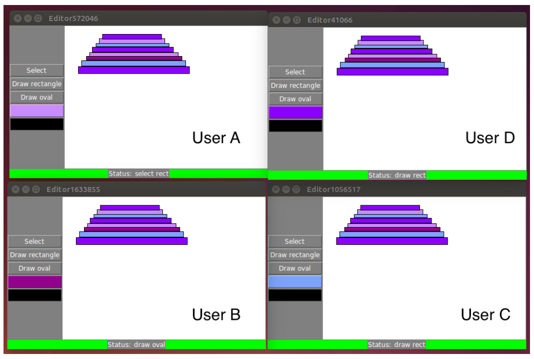
Figure 11.4 – DeTransDraw facing Congestion: Suppose user A, B, C and D (as annotated), where B and C have intermittent connectivity (periodic slow-down). User D due to suspecting its successor on the overlay, C, faces a transient transition to *liquid* phase, whereas C has become isolated (Figure 11.4b, in this snapshot B is in *solid* phase as it is in a period during which its connectivity has been restored). Both B and C have missed some activity updates of A and D (Figure 11.4b). As the connectivity of B and C are restored, canvas consistency and all other functionalities are retrieved.



(a) 4 instances of DeTransDraw



(b) User B is partitioned-away



(c) After the Network Partition ceases

Figure 11.5 – DeTransDraw facing Network Partitioning: Suppose user A, B, C and D (as annotated), where B is partitioned away due to partitioning in the physical network. During the partition the canvas across the partitions diverges with time (see Figure 11.5b); however as the network partition ceases, the application, being self-adaptive and self-optimized, based on the observed phase transitions, does synchronization with the global state, eventually achieves canvas-consistency and full functionality (Figure 11.5c).

Chapter 12

Antidote: Applying Reversibility to a Second Use-Case

In this chapter we apply our concept of *Reversibility* to a second use-case, *Antidote* [25]. *Antidote* is a geo-replicated transactional *CRDT (Conflict Free Replicated Data Type)* key-value store. CRDTs are specially designed data structures which achieve strong eventual consistency and monotonicity (i.e, no rollback) [141]. Antidote is written in the ErLang/OTP programming language. The keys are distributed across physical servers using Riak-Core [142], which hosts a ring-like distributed hash table (DHT) and partitions keys using consistent hashing. In this chapter we briefly describe Antidote and the ring overlay of riak-core. Next, we describe reversibility of this system and our extensions to establish weak reversibility of Antidote against churn.

12.1 Antidote

Geo-scale applications and their underlying data stores depend on distributed infrastructures provided by cloud services in data centers all around the world. The goal is to reduce latency and availability while interacting with clients. To satisfy these objectives, these data stores replicates data both within and across data centers. As shown in literature [143], [144], [145], such setting of data replication is able to provide weaker consistency semantics than classical strong synchronization schemes. Though, CRDTs provide support for shared, mutable and replicated data under eventual consistency, they are not enough to guarantee application correctness. The reason is that CRDTs provide causal consistency guarantee only for the updates to one object, there is no cross-object guarantees. In an effort to solve this problem, *Antidote* [25], [146] provides a common experimental platform, which uses CRDTs with a small number of replicas in a *controllable, confined, and stable execution environment* such as a cloud at the server side. Antidote provides consistent, stable snapshots and atomic multi-CRDT updates, as a result the transactional guarantees provided by Antidote have *Causal+ Consistency* [147], [148] semantics. Causal+ consistency ensures that the causal ordering of operations is respected. It is an optimal spot in the availability-consistency trade-off and the strongest model compatible with availabil-

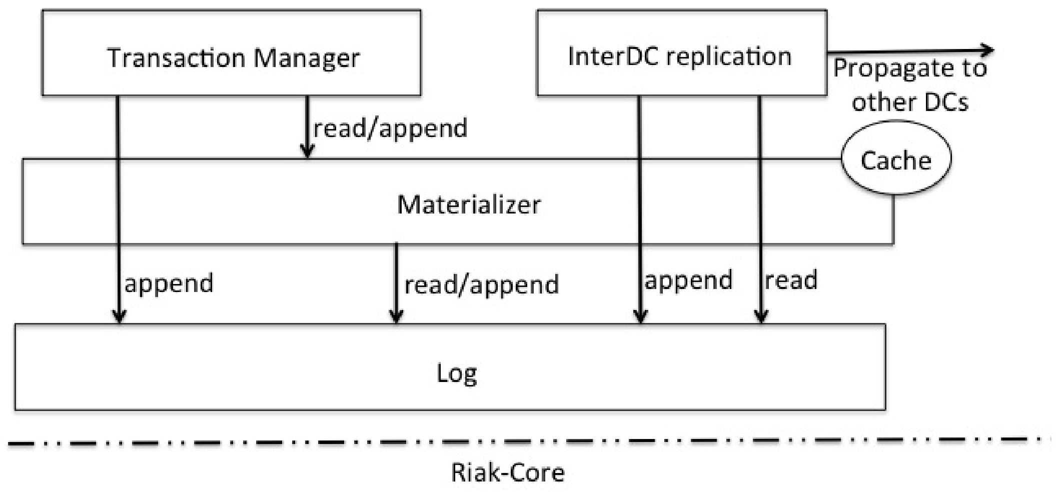


Figure 12.1 – Architecture of Antidote

ity [149] for individual operations. Antidote has a layered architecture, as shown in Figure 12.1. We present a brief description of each layer.

12.1.1 System Settings

Antidote provides a multi-versioned key-value CRDT data store, which is built on top op Riak-Core [142]. At each data center Antidote uses riak-core to partition the set of keys on different nodes. Riak-core is an ErLang OTP application to build distributed, scalable, fault-tolerant applications. Riak-core provides a number of services, namely node liveness and membership, partitioning and distributing work, and managing cluster state. The functionalities of Antidote on each partition, e.g., log, transaction execution are implemented as a virtual node (vnode) provided by riak-core. The current implementation of Antidote [25] uses the same static partitioning scheme at each of the D data centers, where D is in the order to tens.

12.1.2 Log Layer

The Log layer is the foundation of Antidote's architecture. At each data center a log is maintained at each partition. A log is basically a sequence of operations, where each operation contains an operation type and payload. The payload depends on the type of an operation, e.g., update to some CRDT objects, commit or abort of a transaction. An operation also contains meta-data which is used by other layers to obtain correct and consistent objects. Using log-based back-end Antidote provides fast and fault-tolerant write access and efficient management of multi-versioning for CRDT objects. The current implementation of Antidote [25] writes log entries synchronously to hard disk, which is simple, but not necessarily safe and fault-tolerant.

12.1.3 Materializer Layer

The Materializer layer generates snapshots of objects after applying operations. Also, it avoids accessing log for each read operations, thus reducing latency. In order to achieve these objectives all high-layer operations pass though this layer and the materializer process at each partition runs with an in-memory cache of key-value store. The materialization process builds CRDT state after interpreting and combining the payload of operations stored in the log, according to the consistency information encoded in the meta-data of read request parameters. When an update operation is received, the materializer forwards it to the log layer. After receiving acknowledgement from the log layer, the materializer forwards the acknowledgement to the client and stores the operations to its cache, no materialization process is triggered at this stage. When a read request is received, the materializer checks for suitable materialized views (snapshot) and pending operations in the cache. If so, then a new snapshot is created after applying the pending operations to the corresponding snapshot. If no snapshot is found satisfying the read request parameters, the materializer reads missing entries from the log and generates a corresponding snapshot. In order to keep the size of the cache bounded, a separate, concurrently running garbage collector process is integrated into the materializer layer. If the number of snapshots stored for a key int he cache reaches the $snapshot_{threshold}$, the materializer keeps the latest $snapshot_{min}$ snapshots. As the snapshot generation is conducted during reads, this garbage collection mechanism is triggered only by read operations. All operations, except those that are not included in the remaining snapshots and are more recent than the oldest one, are removed. The materializer triggers a read operation if the number of operations stored int he cache exceeds $ops_{threshold}$, to generate a snapshot and remove the operations from the cache that are included in the snapshot.

12.1.4 Transaction Layer

Antidote provides transactional interface to client with Causal+ Consistency semantics, i.e., the transactions have three properties: atomicity (i.e, updates in a transaction are applied in an all-or-none manners), isolation (i.e., intermediate results of a transaction are not visible by others) and mergeable transactions (i.e., results can be merged from two concurrent transactions on the same key in different data centers). A transaction T consists of arbitrary number of reads and writes to multiple keys. T executes on consistent snapshots of all objects accessed. A *read* operation, r, in T includes all updates in T's snapshot and all updates in T which precedes r. A *write* is an update to a key, which is not visible until T is committed. A transaction coordinator (TC) is responsible for executing a transaction T in a data center. A client contacts any node in a data center and starts TC. The client issues *read* and *update* operations via TC.

12.1.5 InterDC Replication

This layer replicates every successfully committed transaction from local data center to other data centers and from remote data centers into the local one. As all these depend on transactional semantics, this module is coupled with the transaction manager. In this thesis, we investigate intra-data center reversibility only, so this layer is ignored for the rest of this work.

12.2 Evaluation of Reversibility

In this section we evaluate reversibility of Antidote (version $0.1.0. - alpha$) against churn (i.e., nodes failing and being replaced by new correct nodes). In this thesis, we are only concerned about reversibility within a data center. In order to measure reversibility we use *throughput* of the system as the metric. Suppose, Antidote running on n servers (i.e., physical nodes) within a data center, providing throughout of N operations per second. During time t to $t+T$, one server crashes and is being replaced by a new server. We evaluate whether the system, after time $t + T$, eventually achieves throughput of N operations per second and also for that, what properties the maintenance of the system should possess.

Antidote is designed for a *controllable, confined, and stable execution environment*. In this chapter, we apply our concept of reversibility to this system. The motivation is to ensure the system's survival in more dynamic environment, e.g., community networks, edge networks. This can open new venues for applications. Also, this will serve as a case study of extending a data-center based system to operate in unreliable and heterogeneous operating environments, thus preparing the way of the topological migration from data-center based computing towards more general edge-computing. In the existing implementation of Antidote [25], membership changes are triggered manually. Once a node crash is declared by a system administrator, the current maintenance in the system uses the gossip protocol to reshuffle the partitions among existing nodes inside a data center. If a node fails, no other node will ever be able to join unless the failed node is removed from the ring and its partitions are reshuffled across alive nodes.

In order to evaluate reversibility of Antidote, we have used a benchmark framework, $basho\_bench$ [150]. Two instances of $basho\_bench$ are used, where each benchmark uses 10 working threads. The workload is evenly distributed among Antidote instances. Each benchmark instance accesses 2000 keys with Pareto distribution and issues read and write operations to Antidote. A read transaction reads a single key and a write transaction updates a single key. The ratio of read and write transactions used is $9 : 1$. We have run all our experiments on Amazon EC2 instances. A cluster of three instances is created, i.e., three physical nodes within a data center. Each instance of Antidote and $basho\_bench$ runs on an EC2 m4.large instance. The number of logical partitions of the identifier space used is 16, i.e., we have used 16 virtual nodes that are evenly distributed among 3 physical nodes. We have run our experiments for 5 minutes and aggregated the results of two benchmark instances. Figure 12.2 and Figure 12.3 show the aggregated throughput and latency results from two $basho\_bench$ instances, of our cluster of 3 Antidote instances in a data center, when there is no stress in the operating environment.

As we can see in Figure 12.2, throughput starts from 6500 transactions per second and decreases a bit with time. The reason is that we have tuned our experimental setup to identify the maximum workload our cluster can handle; increasing the number of $basho\_bench$ instances or working threads per instance more causes the system to be over-stressed and

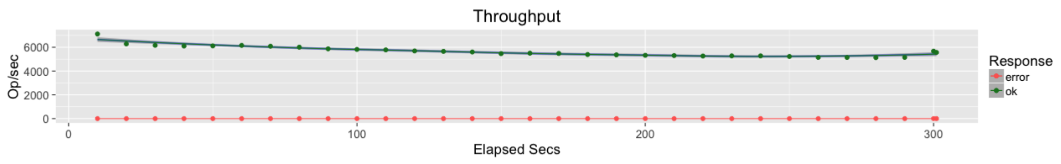


Figure 12.2 – Throughput of 3 Antidote instances over 5 minutes: no stress

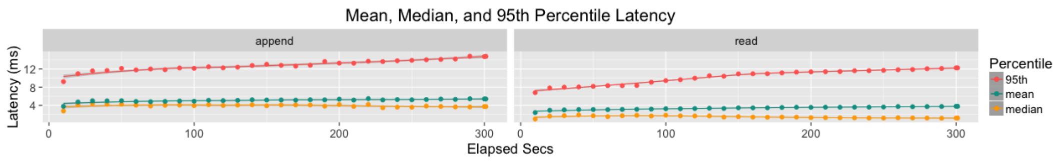


Figure 12.3 – Latency of update and read of Antidote: no stress

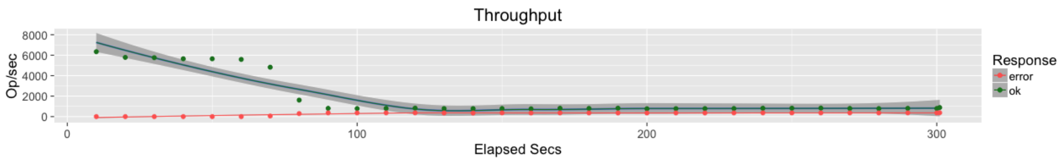


Figure 12.4 – Throughput of 3 Antidote instances over 5 minutes: one instance crashes between 60^{th} and 70^{th} second and a new instance attempts to join between 120^{th} and 130^{th} second. The system is not reversible.

timeouts are triggered. The goal behind identifying the maximum attainable throughput and conducting experiments using that setup is to verify that, when the full capacity of the system is being used for client operations and the system faces stress, e.g., churn, whether the system is able to achieve reversibility as churn recedes. Due to the same reason of running the system on its limits, the mean and 95^{th} percentile latency for append (i.e., update) and read (see Figure 12.3) exhibit a slight increasing trend with time.

In order to assess reversibility of Antidote, we introduce churn as the stress in the operating environment. We have made one Antidote instance (i.e., physical node) crash (we kill the corresponding process using *Pkill* command) and being replaced by a new instance (i.e., a new physical node). As we can see in Figure 12.4, as an instance crashes around 60^{th} second, throughput drops to 1000 transactions per second and remains the same till the end of the experiment. As we have explained before, as membership changes are triggered manually, once a node crashes and is not declared, the new instance fails to join the cluster (the join attempt is made between 120^{th} and 130^{th} second). As a result, the system is unable to recover the throughput-loss, it encounters during stress, as the stress recedes, thus the current implementation of Antidote [25] is not reversible. Due to the same reason, we can see in Figure 12.5, 95^{th} percentile latency for both read and update operations increases to high values (25 millisecond) with time.

The existing maintenance in Antidote requires manual intervention, thus not efficient. In order to combine efficiency with resiliency, we have extended Antidote to trigger maintenance as soon as a membership event is detected. As already explained, Antidote relies on

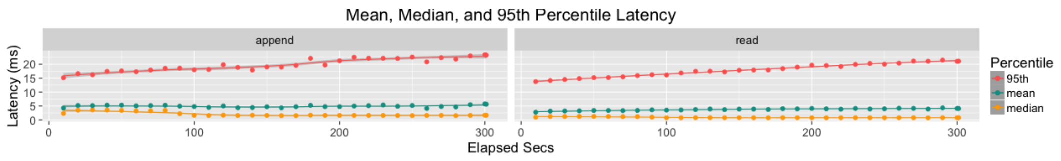


Figure 12.5 – Latency of update and read of Antidote over 5 minutes: one instance crashes between 60th and 70th second and a new instance attempts to join between 120th and 130th second

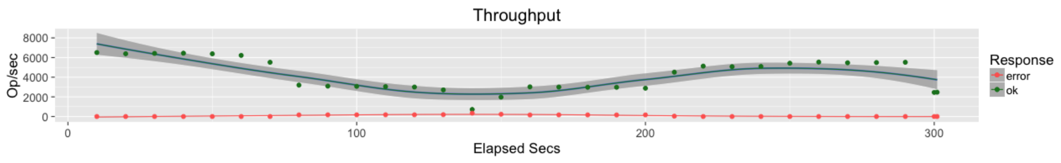


Figure 12.6 – Throughput of 3 Antidote instances over 5 minutes: one instance crashes between 60th and 70th second and a new instance joins the cluster between 130th and 140th second. The system is reversible against these membership change events.

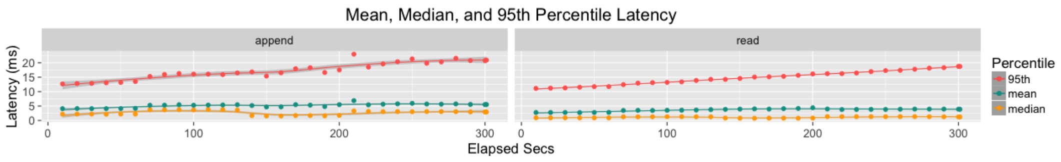


Figure 12.7 – Latency of update and read of Antidote over 5 minutes: one instance crashes between 60th and 70th second and a new instance joins the cluster between 130th and 140th second

riak-core's gossip-based protocol to disseminate membership changes and maintain membership view at each instance. However, the triggering of such maintenance is lazy and requires manual intervention. In order to survive in dynamic operating environments, the maintenance in a system must be efficient, in terms of recovery time and resource usage. For this purpose, we have made the system reactive, whenever a membership change event is detected, the system initiates its gossip-based maintenance. After this extension, the system has achieved reversibility, as shown in Figure 12.6 against membership change events as described before: failure of one Antidote instance and join of a new one. As an Antidote instance crashes between 60th and 70th second, the throughput of the system drops by around $\frac{1}{3}$, becomes 3000 transactions per second. After a new instance joins the cluster around 140th second, the system eventually regains the throughout close to 6000 transactions per second. Figure 12.7 shows the mean, median and 95th percentile latency for append and read transactions. As we can see, these latencies varies (i.e., temporarily increases, followed by a decrease), especially for update transactions, around the instances of time when the membership change events happen.

12.3 Discussion

In this chapter, we have applied our concept of reversibility to a second use-case, Antidote. Antidote is a causal+ consistent transactional key-value store implemented on a consistent hash-based ring overlay. We have investigated the reversibility of this system against churn (i.e., node turnover). As we have observed through our experimentation in a real-world setup, the lazy resilient manually triggered maintenance of this system is unable to achieve reversibility against churn, the maintenance of the system needs to be reactive to accomplish dynamic changes of membership and make the system reversible. This strengthens our conclusion from our first use-case (Beernet) in this thesis that the maintenance in a system needs to be both efficient and resilient to achieve reversibility in stressful operating environment. However, experimentation with Antidote in operating environment having higher churn intensity and other stress elements is subject to future work.

Part V

Conclusions

Chapter 13

Conclusion

The advent and demand for new technologies consistently increases the complexity of distributed systems. The heterogeneity and inclusion of increasing mobility in these systems continues to make the operating environments more inhospitable. In order to build reliable complex computing systems, it is imperative to assess and identify required properties of the maintenance strategy of such systems to achieve complete self-healing and well-defined behavior of the system during the stress, in the entire operating space. In this thesis, we have introduced the concept of *Reversibility*: functionality of a system is a function (aka, the *reversibility function*) of current stress in its operating environment, and is not affected by the failures in the past. Reversibility generalizes standard fault tolerance with nested fault models. When the fault rate goes outside the scope of one model then it is still inside the next model. Reversibility allows opening of new venues for application design, such as mobile and ad hoc networks and Internet of Things, for which existing fault-tolerance techniques are insufficient. As stress is a global condition that cannot be easily measured by individual nodes, we have introduced the concept of *Phase*. In contrast to stress, phase is a per-node property. All nodes in the same phase exhibit the same qualitative properties, which are different for nodes in different phases. Phase at a node gives a qualitative indication of what system operations are available. Each node determines its phase independently and no extra distributed operations are necessary. We approximate the reversibility function using the *phase function*: we say the current set of available operations of the system is a function (aka, the *phase function*) of the current *Phase Configuration*. We define *the Phase Configuration* of the system to be a vector, one entry per node, each entry is the current phase of that particular node.

For the purpose of this work, we have chosen one particular class of complex systems, namely a class of *Structured Overlay Networks (SONs)*, defined by the reference architecture of [26]. To our knowledge, existing literature lacks an assessment or verification of distributed system's reversibility by ensuring complete healing. As a prerequisite of our experimental study, we have organized the entire operating space of distributed systems by identifying the stress sources. We have organized the maintenance principles using a spectrum of *Efficiency* $\leftrightarrow$ *Resiliency*. We have introduced a simple form of resilient maintenance principle, namely *Knowledge Base*. Also, the maintenance principles can be classified along two dimensions: local/global and reactive/proactive. We use a set of strategies which covers all points in this two-dimensional space. As part of making our representative system reversible, we have proposed and evaluated an eventually perfect QoS-aware self-adapting failure detection algorithm and a lazy data migration protocol to fill up the gap in the replica management.

13.1 Stress Investigation

In this thesis, we have investigated several stress sources: Churn, Network Partitioning, Network Dynamicity, and how we can establish reversibility of a system against them. Furthermore, we have investigated the interaction between two stress elements: churn and network partitioning. We have also conducted comparative analysis of the healing capability of maintenance principles, while facing inhospitable environments caused by these stress elements. We have chosen Beernet [13] as a representative system for experimentation. Beernet is a version of Chord [11], the canonical ring-based overlay, except that Beernet has correct lock-free join/leave/failure handling algorithms. We have identified the preconditions, i.e., what properties the maintenance of the system should posses to achieve reversibility for our representative system. Also, the demonstration of a reversible SON against churn, network partitioning and network dynamicity due to packet-loss is presented in this thesis. We have observed that a partition can occur even if there is no communication problem; we also identified and verified the preconditions to ensure partition tolerance for any scenario of network partition. Our results show that both efficient and strongest form of resilient maintenance are required for reversibility, especially when stress (e.g., churn, i.e., nodes failing and being replaced by new correct nodes) perturbs the global state of the system. However, if the stress (e.g., physical network partitioning and dynamicity) only causes change in the local state of nodes, micro-level interactions among nodes (i.e., local corrections at each node) are able to trigger and accomplish macroscopic healing (i.e., merging of multiple overlays and branches to the core-ring). However, the local corrections need to be both proactive and reactive for completion of the recovery process. The result obtained only through such local corrections, without any explicit merge algorithm, is competitive with the one with an explicit overlay merger.

Using the "Stranger model", we have generalized the effect of the interaction between network partition and churn. The stranger model also allows *predicting* when irreversibility arrives (the *cut-off* point) due to *simultaneous* network partition and churn. This can provide useful information for applications, thus support building applications in high-stress environments. We propose to use a Knowledge Base to handle partitions under churn (particularly for high percentage of strangers), that contains knowledge collected in a passive way at each node and injected by an oracle beyond the cut-off point, in order to achieve reversibility. Our results show that an unrestricted or the strongest form of resilient maintenance, e.g., proactive merger using Knowledge Base, is needed for execution during a network partition with churn; however it is not needed for partition repair if the percentage of strangers is low (i.e., either the intensity of churn is low or the duration of partition is short); in that case local maintenance is sufficient to achieve reversibility. Reactive merger using Passive List [37] is never better, and it is superseded by its proactive version using Knowledge Base.

Summary To summarize our empirical study of the effect of various stress sources on a non-trivial complex computing system, in particular a structured overlay network, we draw following conclusions:

- If the stress perturbs the global state of a system, e.g., churn, a strong form of global and resilient maintenance (e.g., gossiping) is required to achieve Reversibility; because such stress erases the states of nodes;
- If the stress causes only local modifications (i.e., only local states of nodes are altered), e.g., network partitioning without churn, packet loss, local maintenance (both reactive and proactive) is enough to achieve Reversibility; because during the stress, nodes' states are preserved, the stress only disrupts the system temporarily (until repair);
- If the stress perturbs both the global state of the system and local states of nodes, e.g., simultaneous network partitioning and churn, there exists a boundary (Cut-Off point) beyond which the system is unable to achieve reversibility by itself.

The outcomes of our case study of stress investigation can be generalized for other distributed system architectures as well. Also, based on this conclusion, we can classify the stress sources of a system in terms of their impacts: global and local. We conjecture that the above conclusion, in terms of the effect of various stress on the local and global state of a system, will hold for any distributed system.

13.2 Practical Applications

In order to build practical applications on top of a reversible system, we propose an efficient approach to compute the phase function. Each node computes the phase function locally based on a simple assumption: the rest of the system has the same qualitative property as the current node. The reason behind such simplification is to avoid any extra distributed operation or bandwidth consumption. In this thesis, we have applied this approach for our representative system. Each node computes its phase and exposes it to the application running on top. Thus, the application running on top of a node, gets qualitative indication of operations currently available in the system and manage their behaviors predictably in stressful environments. In order to achieve this objective, first we have identified different phases and sub-phases, a node, in our representative system, can be in and we have described the semantics of each phase and sub-phase in terms of available operations. We experimentally show that a reversible system does reversible phase transitions, i.e., it "boils" to the gaseous state (becomes disconnected) when the stress increases and "condenses" from gaseous back to solid phase as the stress intensity goes down. We also identify the apparent "critical points" (when more than one phase exists simultaneously in significant fractions of a system) from the experiments while doing such transitions. We have also conducted an experimental study to draw an analogy in our context, of the feedback loop between the thermodynamic system and its environment. As we have observed through experimentation that the system (reversible beernet) and its environment (the physical network) work on each other, when there exists a "heat reservoir" in the environment, which is lossy physical network (i.e., the messages are dropped while en route, thus having infinite message delivery time) in our case. Both the environment and the system affect each others' entropy (i.e., the lack of order or predictability); however eventually the system reaches to a steady-state, i.e., the ratio of nodes in different phases and sub-phases, on an average, remains the same with time.

In this thesis, we have proposed an approach for designing applications for stressful operating environments. We have applied the concept of reversibility in the context of application design. We extend each node of the underlying system to be self-aware and demonstrate how a self-aware node can determine its phase without any global synchronization. We have presented an API to make the phase of a node explicit to the application layer and analyze the applications of our concepts of phase and phase transitions toward designing predictable and reversible systems. We have described phase-aware design of a use-case application. Using this use-case application, we exhibit, based on the phase and phase transitions of the underlying node, how an application can achieve reversibility in the application-level semantics. Furthermore, the application can have improved behavior from the viewpoint of the user, i.e.: have self adaptation, self optimization, and, be behaviorally predictable to the user in high-stress operating environments. Thus, in this thesis, we propose a principled way to build applications that survive in stressful operating environments by using the concepts of *reversibility* and *phase*. Using case studies we have shown the usefulness of these concepts for building large-scale long-running Internet applications in highly dynamic operating environments, such as edge networks.

Summary In our case studies we have applied the concept of phase to our representative system. The goal was to approximate the available functionality of the system at each node. For this purpose, we have identified the qualitative properties that drives the system functionality and identified different phases in our system. Next, we have verified that if there is stress in the local environment, each node faces phase transitions. As we have extended our system to be reversible, with varying stress we have observed reversible phase transitions at system-level. We give qualitative indications to the applications regarding available functionality of the system using APIs and use it for phase-aware application design. Using a case study we show how an application can be extended to have predictable behavior and achieve reversibility in the application-level semantics.

Our approach of designing phase-aware applications can be extended to other system architectures and applications. In order to apply our approach of approximating system functionality at each component, we need to identify different observable phases in a system based on the qualitative properties of each component that have effects on the global behavior of the system. Also, as using phase we are encapsulating the impact of stress, it is essential to verify that if there is stress in the local environment of a component, it changes phase independently. Each component can compute the phase function locally either based on its own phase solely (as is done in our case study) or by maintaining a phase base as outlined in Section 10.1, based on application-level policy. Our phase APIs can be easily extended so that an application can provide directions (e.g., 0 = compute the phase function based on a node's phase only, 1 = compute the phase function based on a node's and its neighbors phases and 2 = compute the phase function based on the phase base) regarding the quality of approximation, based on application-specific cost vs. quality policy. The application can use the information of approximated available functionality of the underlying system as per application-level semantics, e.g., enhancing self-management properties and achieve reversibility, reducing the probability of loss of critical data, improving usability etc.

Chapter 14

Future Work

To conclude this thesis, in this chapter, we discuss some research ideas derived from the results obtained so far. Some of these ideas are continuation of the work presented in this thesis, others explore the other directions of investigations, not addressed in this thesis. However, all of these ideas are relevant to building *Reversible* and *Predictable* distributed systems and application services.

14.1 Formal Analysis of Reversibility

In this thesis, we have followed only empirical approach to investigate the design and behavior of distributed systems that can survive, yet able to provide useful functionality in stressful operating environments. As one outcome of this investigation, we have proposed the concept of *Reversibility*. We have experimentally evaluated and established reversibility for our representative systems. However, no theoretical analysis or verification of reversibility has been presented in this thesis. Like any scientific concept, the next step to take this concept forward, is to conduct formal analysis (e.g., theorem proving) and verification (e.g., using model checking) of reversibility. This kind of theoretical research will strengthen the concept and establish reversibility as a desirable property for any system, operating in any operating condition.

14.2 Better Approximation of the Reversibility Function

In this thesis, we have proposed an approach to approximate the set of available operations (the reversibility function) in a system using the concept of *Phase* (see Chapter 2). A Phase is a subset of a system for which the qualitative properties are essentially the same. We have defined phase configuration as a vector, one entry per node, and approximated the reversibility function using the phase function. However, we have computed the phase function for our representative system using a simplifying assumption to avoid any extra distributed operation or resource consumption. We computed the phase function at each node assuming that the other nodes in the system have the same qualitative properties as the current node. However, as we have mentioned (see Chapter 10), we can have a better approximation of the reversibility function by maintaining a *Phase Base (PB)* at each node. The phase base is a local view of phase configuration. Each node can communicate its PB with others to enhance its view. This can be efficiently done by piggybacking PB with messages routed via each node, to avoid any extra distributed operations (though extra bandwidth consumption is unavoidable), or using a stronger algorithm, such as gossip, where each node asks a random node to send its PB that it merges with its own PB to converge to the current phase configuration of the system. The computation of phase function, taking PB as parameter, can be a sophisticated algorithm, e.g., machine learning, to have a better approximation of available functionality of the system. A study which implements these ideas and investigates extra resource consumption due to such extensions, to justify the worthiness of such algorithms, is required to identify the optimal way to approximate the reversibility function for practical application design.

14.3 Exploring the Thermodynamic Analogy

In this thesis, we have presented our first investigation about phase and phase transitions in a SON with churn and network dynamicity due to packet-loss. There is a strong analogy between SON maintenance in the face of environment stress, and thermodynamics of physical systems. This analogy shows up best in the packet loss experiments in Section 10.4, since in these experiments the environment affects the system and the system affects the environment (bidirectional influence). In the simulations done for other experiments (see Section 10.3), we do not model these bidirectional influences, so the thermodynamic analogy is not as strong.

When the system and its environment work on each other due to presence of a stress ("heat source" in thermodynamic terminology) in the environment , we have observed that our representative system, under stress (packet-loss in the physical network), has exhibited the same properties as a thermodynamic system (see Section 10.4). We intend to investigate further about this analogy and also the analogy between phase in SONs and in physical systems. Furthermore, defining entropy in our context and investigation about applying the laws of thermodynamics in computing system design is a promising research direction. Formal analysis and empirical investigation of such analogy and multi-disciplinary research will enrich our field, help designing well-defined and sustainable systems.

14.4 Maintenance Principles and Efficient Self-Healing

In future work, we intend to improve the maintenance principles, e.g., we intend to use gossip protocols to improve the knowledge base technique. In order to use this technique in practical systems, it is required to implement an algorithm for garbage collection of node information, since it is not possible to collect information on new nodes indefinitely. Because of churn, the number of new nodes seen grows without bounds. This problem is studied in peer sampling services [151, 152, 153]. Also, in our experiments, we have introduced some parameters, e.g., *Join Timeout* for the new peers to issue repeated join requests, we intend to improve these parameters, which are currently empirical. Besides, it is necessary for efficient survival/self- healing in an inhospitable environment to adapt such system parameters to the operating environment. This also requires a study to identify the influences of different parameters on convergence time of healing or phase transitions/critical points.

14.5 Software Rejuvenation

Software rejuvenation [154, 155, 156] consists of restarting the system from a subset of its execution state. During the restart, the system will rebuild its state from this subset. This solves many problems of long-running systems related to increasing data corruption and inflated memory usage. An analogous approach is ubiquitous in biological organisms: almost all organisms grow old and die eventually, and child organisms start with a new clean state. Software rejuvenation as maintenance technique is not studied in this thesis.

14.6 Limits of Reversibility and Additional Stress Sources

In this thesis we have explored three stress sources individually and designed the maintenance of a system such that the system attains weak reversibility against them. We have also investigated the interaction between two stress sources (Chapter 8): churn and network partitioning, and found a cut-off point in time during such interaction, beyond which the system becomes irreversible. The only way to go beyond the cut-off point is for the system to receive information from its environment (from an oracle). Two stress sources that were not studied in this thesis are workload and system scale. When the workload increases, the system throughput will approach an asymptote and the system latency will increase. We suspect that this will reduce the effectiveness of the maintenance strategies. We also suspect that increasing system scale might create challenges for the maintenance of the system, however, from thermodynamic point of view, the system may become more stable with increasing node count [47]. In the multidimensional space of stress sources (churn, partitioning, workload, communications delay and packet loss, etc.), there is a “boundary” beyond which reversibility is no longer possible. For example, we have shown that there exists a cut-off point when churn and partitioning are combined. Future work can study such boundary in other parts of the space of stress sources. Furthermore, we fully expect to encounter new and unexpected phenomena when studying such interactions, as was the case during the research of this thesis.

14.7 Data-Level Reversibility

In this thesis, we have studied reversibility only for “connectivity”, which is the lower layer of a structured overlay network. We have not studied reversibility for replicated storage. With high stress, especially stress which erases nodes’ states (e.g., churn), data is lost since node data is lost due to rapid turnover. At some point, the replicated storage loses data and therefore some storage operations are no longer possible. Future work can study reversibility with respect to storage.

14.8 Experimentation and Validation on Real-World Environment

In future work, we intend to evaluate our system in a real-world (non-simulated) dynamic environment, like PlanetLab [157] and Community Networks, to verify the findings from the simulation. Also, we intend to investigate other application design aspects based on the phase concept that can further enhance the behavior of distributed applications, and experiment with applications in large-scale and real-live stressful environments.

Bibliography

[1] Richard John Anthony. Emergence: A paradigm for robust and scalable distributed applications. In *Proceedings of IEEE International Conference on Autonomic Computing (ICAC'04)*, pages 132–139, 2004.

[2] T. De Wolf, G. Samaey, T. Holvoet, and D. Roose. Decentralised autonomic computing: Analysing self-organising emergent behaviour using advanced numerical methods. In *Autonomic Computing, 2005. ICAC 2005. Proceedings. Second International Conference on*, pages 52–63, June 2005.

[3] Ozalp Babaoglu, Márk Jelasity, and Alberto Montresor. Grassroots approach to self-management in large-scale distributed systems. In Jean-Pierre Banâtre, Pascal Fradet, Jean-Louis Giavitto, and Olivier Michel, editors, *Unconventional Programming Paradigms: International Workshop UPP 2004, Le Mont Saint Michel, France, September 15-17, 2004, Revised Selected and Invited Papers*, pages 286–296. Springer Berlin Heidelberg, Berlin, Heidelberg, 2005.

[4] Napster. Inc. napster. `http://www.napster.com`, 1999. Accessed: 2016.

[5] Audiogalaxy. The Audiogalaxy satellite. `http://www.audiogalaxy.com/`, 2001. Accessed: 2004.

[6] OpenNap Community. Open source napster server. `http://opennap.sourceforge.net/`, 2001. Accessed: 2016.

[7] Gnutella. Gnutella. `http://www.gnutella.com/`, 2006.

[8] FreeNet Community. The freenet project. `http://freenetproject.org`, 2003. Accessed: 2016.

[9] K. Gummadi, R. Gummadi, S. Gribble, S. Ratnasamy, S. Shenker, and I. Stoica. The impact of dht routing geometry on resilience and proximity. In *Proceedings of the 2003 Conference on Applications, Technologies, Architectures, and Protocols for Computer Communications*, SIGCOMM '03, pages 381–394, New York, NY, USA, 2003. ACM.

[10] Emma Brunskill. Building peer-to-peer systems with chord, a distributed lookup service. In *Proceedings of the Eighth Workshop on Hot Topics in Operating Systems*, HOTOS '01, pages 81–, Washington, DC, USA, 2001. IEEE Computer Society.

[11] Ion Stoica, Robert Morris, David Karger, M. Frans Kaashoek, and Hari Balakrishnan. Chord: A scalable peer-to-peer lookup service for internet applications. In *Proceedings of the 2001 Conference on Applications, Technologies, Architectures, and Protocols for Computer Communications*, SIGCOMM '01, pages 149–160, New York, NY, USA, 2001. ACM.

[12] Luc Onana Alima, Sameh El-ansary, Per Brand, and Seif Haridi. DKS(n,k,f): a family of low communication, scalable and fault-tolerant infrastructures for p2p applications. In *Proceedings of the 3rd IEEE/ACM International Symposium on Cluster Computing and the GRID (CCGRID)*, pages 344–350, 2003.

[13] Boris Mejías Candia. *Beernet: A Relaxed Approach to the Design of Scalable Systems with Self-Managing Behaviour and Transactional Robust Storage*. PhD thesis, ICTEAM, Université catholique de Louvain, Louvain-la-Neuve, Belgium, October 2010.

[14] Boris Mejías and Peter Van Roy. Beernet: Building self-managing decentralized systems with replicated transactional storage. *IJARAS: International Journal of Adaptive, Resilient and Automatic Systems*, July 2010.

[15] Jeffrey O. Kephart and David M. Chess. The vision of autonomic computing. *Computer*, 36(1):41–50, January 2003.

[16] Felix C. Gärtner. Fundamentals of fault-tolerant distributed computing in asynchronous environments. *ACM Comput. Surv.*, 31(1):1–26, March 1999.

[17] Abhishek Dhama. *A compositional framework for Designing self-stabilizing distributed algorithms*. PhD thesis, Universität Oldenburg, Oldenburg, Germany, 2013.

[18] Marco Schneider. Self-stabilization. *ACM Computing Surveys*, 25(1):45–67, March 1993.

[19] Edsger W. Dijkstra. Self-stabilizing systems in spite of distributed control. *Communications of the ACM*, 17(11):643–644, November 1974.

[20] A. Arora and S. S. Kulkarni. Component based design of multitolerant systems. *IEEE Transactions on Software Engineering*, 24(1):63–78, Jan 1998.

[21] Ken Birman, Coimbatore Ch, Danny Dolev, and Robbert Van Renesse. How the hidden hand shapes the market for software reliability. In *IEEE Workshop on Applied Software Reliability, Philadelphia, PA, USA, June, 2006. Proceedings*, June 2006.

[22] J. Dean. Designs, lessons and advice from building large distributed systems. *Keynote from LADIS*, 2009.

[23] Ken Birman, Gregory Chockler, and Robbert van Renesse. Toward a cloud computing research agenda. *SIGACT News*, 40(2):68–80, June 2009.

[24] Ian Wilkes. What second life can teach your datacenter about scaling web apps. *Ars Technica*, February 2010. Accessed: 2016.

[25] SyncFree Consortium. Antidote reference platform. `http://github.com/SyncFree/antidote`, 2015. Accessed:2016.

[26] K. Aberer, L. O. Alima, A. Ghodsi, S. Girdzijauskas, S. Haridi, and M. Hauswirth. The essence of P2P: a reference architecture for overlay networks. In *Proceedings of the Fifth IEEE International Conference on Peer-to-Peer Computing (P2P'05)*, pages 11–20, Aug 2005.

[27] Jean Bacon and Ken Moody. Consistency in distributed systems. `https://www.cl.cam.ac.uk/teaching/0910/ConcDistS/11a-cons-tx.pdf`, 2010. Accessed: 2016.

[28] Werner Vogels. Eventually consistent. *Commun. ACM*, 52(1):40–44, January 2009.

[29] Miguel Castro, Manuel Costa, and Antony Rowstron. Performance and dependability of structured peer-to-peer overlays. In *Proceedings of the 2004 International Conference on Dependable Systems and Networks*, DSN '04, pages 9–, Washington, DC, USA, 2004. IEEE Computer Society.

[30] Tallat M. Shafaat, Monika Moser, Thorsten Schütt, Alexander Reinefeld, Ali Ghodsi, and Seif Haridi. Key-based consistency and availability in structured overlay networks. In *Proceedings of the 3rd International Conference on Scalable Information Systems*, InfoScale '08, pages 13:1–13:5, ICST, Brussels, Belgium, Belgium, 2008. ICST (Institute for Computer Sciences, Social-Informatics and Telecommunications Engineering).

[31] Supriya Krishnamurthy, Sameh El-Ansary, Erik Aurell, and Seif Haridi. An analytical study of a structured overlay in the presence of dynamic membership. *IEEE/ACM Transactions on Networking (TON)*, 16(4):814–825, August 2008.

[32] Supriya Krishnamurthy, Sameh El-Ansary, Erik Aurell, and Seif Haridi. A statistical theory of chord under churn. In *Proceedings of the 4th International Conference on Peer-to-Peer Systems*, IPTPS'05, pages 93–103, Berlin, Heidelberg, 2005. Springer-Verlag.

[33] Supriya Krishnamurthy, Sameh El-Ansary, Erik Aurell, and Seif Haridi. Comparing maintenance strategies for overlays. In *Proceedings of the 16th Euromicro Conference on Parallel, Distributed and Network-Based Processing (PDP 2008)*, PDP '08, pages 473–482, Washington, DC, USA, 2008. IEEE Computer Society.

[34] David Liben-Nowell, Hari Balakrishnan, and David Karger. Analysis of the evolution of peer-to-peer systems. In *Proceedings of the Twenty-first Annual Symposium on Principles of Distributed Computing*, PODC '02, pages 233–242, New York, NY, USA, 2002. ACM.

[35] Joseph S. Kong, Jesse S.A. Bridgewater, and Vwani P. Roychowdhury. Resilience of structured P2P systems under churn: The reachable component method. *Computer Communications*, 31(10):2109–2123, June 2008.

[36] Supriya Krishnamurthy and John Ardelius. An analytical framework for the performance evaluation of proximity-aware structured overlays. Technical report, Swedish Institute of Computer Science (SICS), 2008.

[37] Tallat M. Shafat. *Partition Tolerance and Data Consistency in Structured Overlay Networks*. PhD thesis, KTH Royal Institute of Technology, Sweden, 2013.

[38] G. Kunzmann and A. Binzenhofer. Autonomically improving the security and robustness of structured P2P overlays. In *Systems and Networks Communications, 2006. ICSNC '06. International Conference on*, pages 18–18, Oct 2006.

[39] X. Xiang. Coping with structured P2P network partitions and unifications. *JCIT: Journal of Convergence Information Technology*, 6(4):25–33, 2011.

[40] X. Xiang. Merging and splitting structured peer-to-peer systems. In *2008 International Symposium on Information Science and Engineering*, volume 1, pages 501–505, Dec 2008.

[41] Anwitaman Datta. Merging intra-planetary index structures: Decentralized bootstrapping of overlays. In *Proceedings of SASO 2007, IEEE International Conference on Self-Adaptive and Self-Organizing Systems*, 2007.

[42] Anwitaman Datta and Karl Aberer. The challenges of merging two similar structured overlays: A tale of two networks. In *Proceedings of the First International Conference, and Proceedings of the Third International Conference on New Trends in Network Architectures and Services Conference on Self-Organising Systems*, IWSOS'06/EuroNGI'06, pages 7–22, Berlin, Heidelberg, 2006. Springer-Verlag.

[43] B. Y. Zhao, Ling Huang, J. Stribling, A. D. Joseph, and J. D. Kubiatowicz. Exploiting routing redundancy via structured peer-to-peer overlays. In *Network Protocols, 2003. Proceedings. 11th IEEE International Conference on*, pages 246–257, Nov 2003.

[44] György Bognár Zoltán Czirkos and Gábor Hosszú. Packet loss and overlay size aware broadcast in the kademlia p2p system. *ACEEE International Journal of Communications*, 4(1):8, july 2013.

[45] S. El-Ansary, E. Aurell, P. Brand, and S. Haridi. Experience with a physics-style approach for the study of self properties in structured overlay networks. In *Proceedings of SELF-STAR: International Workshop on Self-* Properties in Complex Information Systems*, May 2004.

[46] Sameh El-Ansary, Erik Aurell, and Seif Haridi. A physics-inspired performance evaluation of a structured peer-to-peer overlay network. In *Proceedings of the IASTED International Conference on Parallel and Distributed Computing and Networks, part of the 23rd Multi-Conference on Applied Informatics, Innsbruck, Austria, February 15-17, 2005*, pages 116–122, 2005.

[47] John Kubiatowicz. Extracting guarantees from chaos. *Commun. ACM*, 46(2):33–38, February 2003.

[48] George Fletcher. Decentralized object location in dynamic peer-to-peer distributed systems. In *Project 3*, B649, Dr. Plale, July 2003.

[49] Ingo Scholtes and Juan Claudio Tessone. Organic design of massively distributed systems: A complex networks perspective. *Informatik-Spektrum*, 35(2):75–86, 2012.

[50] Ruma R. Paul, Peter Van Roy, and Vladimir Vlassov. Reversible phase transitions in a structured overlay network with churn. In *Networked Systems (NETYS), 2016 4th International Conference on*, May 2016.

[51] Wikipedia. Matryoshka doll. `https://en.wikipedia.org/wiki/Matryoshka_doll`, August 2016. Accessed: 2016.

[52] Wikipedia. Phase (matter). `https://en.wikipedia.org/wiki/Phase_(matter)`, July 2016. Accessed: 2016.

[53] Thorsten Schütt, Florian Schintke, and Alexander Reinefeld. A structured overlay for multi-dimensional range queries. In *Proceedings of the 13th International Euro-Par Conference on Parallel Processing*, Euro-Par'07, pages 503–513, Berlin, Heidelberg, 2007. Springer-Verlag.

[54] Nicholas J. A. Harvey, Michael B. Jones, Stefan Saroiu, Marvin Theimer, and Alec Wolman. Skipnet: A scalable overlay network with practical locality properties. In *Proceedings of the 4th Conference on USENIX Symposium on Internet Technologies and Systems - Volume 4*, USITS'03, pages 9–9, Berkeley, CA, USA, 2003. USENIX Association.

[55] M. Frans Kaashoek and David R. Karger. Koorde: A simple degree-optimal distributed hash table. In M. Frans Kaashoek and Ion Stoica, editors, *Peer-to-Peer Systems II: Second International Workshop, IPTPS 2003, Berkeley, CA, USA, February 21-22, 2003. Revised Papers*, pages 98–107. Springer Berlin Heidelberg, Berlin, Heidelberg, 2003.

[56] Valentin Mesaros, Bruno Carton, and Peter Van Roy. P2PS: Peer-to-peer development platform for mozart. In Peter Van Roy, editor, *Multiparadigm Programming in Mozart/Oz*, volume 3389 of *Lecture Notes in Computer Science*, pages 125–136. Springer Berlin Heidelberg, 2005.

[57] Ashwin R. Bharambe, Mukesh Agrawal, and Srinivasan Seshan. Mercury: Supporting scalable multi-attribute range queries. *SIGCOMM Computer Communication Review*, 34(4):353–366, August 2004.

[58] Ashwin R. Bharambe, Mukesh Agrawal, and Srinivasan Seshan. Mercury: Supporting scalable multi-attribute range queries. In *Proceedings of the 2004 Conference on Applications, Technologies, Architectures, and Protocols for Computer Communications*, SIGCOMM '04, pages 353–366, New York, NY, USA, 2004. ACM.

[59] B. Leong, B. Liskov, and E. Demaine. EpiChord: Parallelizing the chord lookup algorithm with reactive routing state managements. In *Proceedings of the ACM SIGCOMM 2004 Symposium on Communication, Architecture, and Protocols*, Singapore, November 2004. IEEE Computer Society.

[60] Jinyang Li, Jeremy Stribling, Robert Morris, and M. Frans Kaashoek. Bandwidth-efficient management of dht routing tables. In *Proceedings of the 2Nd Conference on Symposium on Networked Systems Design & Implementation - Volume 2*, NSDI'05, pages 99–114, Berkeley, CA, USA, 2005. USENIX Association.

[61] G. S. Manku, M. Bawa, and P. Raghavan. Symphony: Distributed hashing in a small world. In *Proceedings of the 4th USENIX Symposium on Internet Technologies and Systems (USITS'03)*, Seattle, WA, USA, March 2003. USENIX.

[62] Jon Kleinberg. The small-world phenomenon: An algorithmic perspective. In *Proceedings of the Thirty-second Annual ACM Symposium on Theory of Computing*, STOC '00, pages 163–170, New York, NY, USA, 2000. ACM.

[63] Márk Jelasity and Ozalp Babaoglu. T-man: Gossip-based overlay topology management. In *Proceedings of the Third International Conference on Engineering Self-Organising Systems*, ESOA'05, pages 1–15, Berlin, Heidelberg, 2006. Springer-Verlag.

[64] Márk Jelasity, Alberto Montresor, and Ozalp Babaoglu. T-man: Gossip-based fast overlay topology construction. *Comput. Netw.*, 53(13):2321–2339, August 2009.

[65] João Leitão, José Pereira, and Luís Rodrigues. Epidemic broadcast trees. In *Proceedings of the 26th IEEE International Symposium on Reliable Distributed Systems*, SRDS '07, pages 301–310, Washington, DC, USA, 2007. IEEE Computer Society.

[66] Karl Aberer, Anwitaman Datta, and Manfred Hauswirth. Route maintenance overheads in DHT overlays. In *Workshop on Distributed Data and Structures*, 2003.

[67] Ruma R. Paul, Peter Van Roy, and Vladimir Vlassov. An empirical study of the global behavior of a structured overlay network. In *Peer-to-Peer Computing (P2P), 14-th IEEE International Conference on*, pages 1–5, Sept 2014.

[68] Alan Demers, Dan Greene, Carl Hauser, Wes Irish, John Larson, Scott Shenker, Howard Sturgis, Dan Swinehart, and Doug Terry. Epidemic algorithms for replicated database maintenance. In *Proceedings of the Sixth Annual ACM Symposium on Principles of Distributed Computing*, PODC '87, pages 1–12, New York, NY, USA, 1987. ACM.

[69] Michael J. Freedman, Karthik Lakshminarayanan, Sean Rhea, and Ion Stoica. Non-transitive connectivity and DHTs. In *Proceedings of the 2nd Workshop on Real Large Distributed Systems*, 2005.

[70] Ali Ghodsi. *Distributed k-ary System: Algorithms for Distributed Hash Tables*. PhD thesis, KTH Royal Institute of Technology, Sweden, 2006.

[71] Ranjita Bhagwan, Stefan Savage, and GeoffreyM. Voelker. Understanding availability. In M.Frans Kaashoek and Ion Stoica, editors, *Peer-to-Peer Systems II*, volume 2735 of *Lecture Notes in Computer Science*, pages 256–267. Springer Berlin Heidelberg, 2003.

[72] Stefan Saroiu, P. Krishna Gummadi, and Steven D. Gribble. A measurement study of peer-to-peer file sharing systems. In *Proceedings of the Multimedia Computing and Networking (MMCN)*, San Jose, CA, USA, January 2002.

[73] Daniel Stutzbach and Reza Rejaie. Understanding churn in peer-to-peer networks. In *Proceedings of the 6th ACM SIGCOMM Conference on Internet Measurement*, IMC '06, pages 189–202, New York, NY, USA, 2006. ACM.

[74] M. Steiner, T. En-Najjary, and E.W. Biersack. Long term study of peer behavior in the KAD DHT. *Networking, IEEE/ACM Transactions on*, 17(5):1371–1384, Oct 2009.

[75] Farnam Jahanian, Craig Labovitz, and Abha Ahuja. Experimental study of internet stability and wide-area backbone failures. Technical Report CSE-TR-382-98, University of Michigan, November 1998.

[76] David Oppenheimer, Archana Ganapathi, and David A. Patterson. Why do internet services fail, and what can be done about it? In *Proceedings of the 4th Conference on USENIX Symposium on Internet Technologies and Systems - Volume 4*, USITS'03, pages 1–1, Berkeley, CA, USA, 2003. USENIX Association.

[77] Undersea cable damage 2006. Taiwan earthquake shakes Internet. `http://www.theregister.co.uk/2006/12/27/boxing_day_earthquake_taiwan/`, April 2013. Accessed: 2016.

[78] Vern Paxson. End-to-end routing behavior in the internet. *SIGCOMM Comput. Commun. Rev.*, 36(5):41–56, October 2006.

[79] Taiwan quake exposes internet vulnerability. `http://www.globalsecuritynews.com/Asia/Wolfe-Adam/Taiwan-Quake-Exposes-Internet-Vulnerability`, January 2007. Accessed: 2016.

[80] ISP quarrel partitions Internet. `http://www.wired.com/threatlevel/2008/03/isp-quarrel-par/`, March 2008. Accessed: 2016.

[81] The Cogent-Level 3 Dispute. `http://www.lookingglassnews.org/viewstory.php?storyid=2883`, October 2005. Accessed: 2016.

[82] Andrew McLaughlin. Egypt's big Internet disconnect. `http://www.theguardian.com/commentisfree/2011/jan/31/egypt-internet-uncensored-cutoff-disconnect`, January 2011. Accessed: 2016.

[83] Michael Mitzenmacher and Rajmohan Rajaraman. Towards more complete models of tcp latency and throughput. *J. Supercomput.*, 20(2):137–160, September 2001.

[84] Programming Languages and Distributed Computing Research Group. Beernet: pbeer-to-pbeer network. `http://beernet.info.ucl.ac.be`, 2009. Accessed: 2016.

[85] Christian Cachin, Rachid Guerraoui, and Luis Rodrigues. *Introduction to Reliable and Secure Distributed Programming*. Springer, second edition, 2011.

[86] Wei Chen, Sam Toueg, and Marcos Kawazoe Aguilera. On the quality of service of failure detectors. *IEEE Trans. Comput.*, 51(5):561–580, May 2002.

[87] Jay Aikat, Jasleen Kaur, F. Donelson Smith, and Kevin Jeffay. Variability in TCP round-trip times. In *Proceedings of the 3rd ACM SIGCOMM Conference on Internet Measurement*, IMC '03, pages 279–284, New York, NY, USA, 2003. ACM.

[88] M. Bertier, O. Marin, and P. Sens. Implementation and performance evaluation of an adaptable failure detector. In *Dependable Systems and Networks, 2002. DSN 2002. Proceedings. International Conference on*, pages 354–363, 2002.

[89] Benjamin Satzger, Andreas Pietzowski, Wolfgang Trumler, and Theo Ungerer. A new adaptive accrual failure detector for dependable distributed systems. In *Proceedings of the 2007 ACM Symposium on Applied Computing*, SAC '07, pages 551–555, New York, NY, USA, 2007. ACM.

[90] Network Working Group. RFC 2988 : Computing TCP's retransmission. `http://www.rfc-editor.org/rfc/rfc2988.txt`, 2000. Accessed:2016.

[91] Mikel Larrea, Sergio Arévalo, and Antonio Fernández. Efficient algorithms to implement unreliable failure detectors in partially synchronous systems. In *Proceedings of the 13th International Symposium on Distributed Computing*, pages 34–48, London, UK, UK, 1999. Springer-Verlag.

[92] C. Fetzer, M. Raynal, and F. Tronel. An adaptive failure detection protocol. In *Dependable Computing, 2001. Proceedings. 2001 Pacific Rim International Symposium on*, pages 146–153, 2001.

[93] C. Fetzer, U. Schmid, and M. Susskraut. On the possibility of consensus in asynchronous systems with finite average response times. In *Distributed Computing Systems, 2005. ICDCS 2005. Proceedings. 25th IEEE International Conference on*, pages 271–280, June 2005.

[94] Monika Moser and Seif Haridi. Atomic commitment in transactional dhts. In Thierry Priol and Marco Vanneschi, editors, *Towards Next Generation Grids: Proceedings of the CoreGRID Symposium 2007*, pages 151–161, Boston, MA, 2007. Springer US.

[95] Mozart Consortium. The Mozart-Oz programming system. `http://mozart.github.io/`, 2013. Accessed:2016.

[96] Antony Rowstron and Peter Druschel. Pastry:scalable, distributed object location and routing for large-scale peer-to-peer systems. In *Proceedings of the IFIP/ACM International Conference on Distributed Systems Platforms (Middleware)*, pages 329–350, November 2001.

[97] S. Apel and K. Böhm. Self-organization in overlay networks. In *Proceedings of CAISE Workshop Adaptive and Self-Managing Enterprise Applications*, volume 2, pages 139–153, 2005.

[98] Ruma R. Paul, Peter Van Roy, and Vladimir Vlassov. Interaction between network partitioning and churn in a self-healing structured overlay network. In *Parallel and Distributed Systems (ICPADS), 2015 IEEE 21st International Conference on*, pages 232–241, Dec 2015.

[99] Seth Gilbert and Nancy Lynch. Brewer's conjecture and the feasibility of consistent, available, partition-tolerant web services. *SIGACT News*, 33(2):51–59, June 2002.

[100] Karl Aberer, Philippe Cudré-Mauroux, Anwitaman Datta, Zoran Despotovic, Manfred Hauswirth, Magdalena Punceva, and Roman Schmidt. P-Grid: A self-organizing structured P2P system. *SIGMOD Rec.*, 32(3):29–33, September 2003.

[101] A. Datta. Merging ring-structured overlay indices: toward network-data transparency. *Computing*, 94(8-10):783–809, 2012.

[102] S.M. Das, L.R. Dondeti, V. Narayanan, and R.S. Jayaram. Methods and apparatus for merging peer-to-peer overlay networks, September 2 2014. US Patent 8,825,768.

[103] David Liben-Nowell, Hari Balakrishnan, and David Karger. Observations on the dynamic evolution of peer-to-peer networks. In *Revised Papers from the First International Workshop on Peer-to-Peer Systems*, IPTPS '01, pages 22–33, London, UK, UK, 2002. Springer-Verlag.

[104] Ratul Mahajan, Miguel Castro, and Antony Rowstron. Controlling the cost of reliability in peer-to-peer overlays. In M.Frans Kaashoek and Ion Stoica, editors, *Peer-to-Peer Systems II*, volume 2735 of *Lecture Notes in Computer Science*, pages 21–32. Springer Berlin Heidelberg, 2003.

[105] Martin Ellis, Dimitrios P. Pezaros, Theodore Kypraios, and Colin Perkins. A two-level markov model for packet loss in udp/ip-based real-time video applications targeting residential users. *Comput. Netw.*, 70:384–399, September 2014.

[106] M. S. Borella, D. Swider, S. Uludag, and G. B. Brewster. Internet packet loss: measurement and implications for end-to-end QoS. In *Architectural and OS Support for Multimedia Applications/Flexible Communication Systems/Wireless Networks and Mobile Computing., 1998 Proceedings of the 1998 ICPP Workshops on*, pages 3–12, Aug 1998.

[107] Jean-Chrysostome Bolot. Characterizing end-to-end packet delay and loss in the internet. *J. High Speed Netw.*, 2(3):305–323, July 1993.

[108] Nick Feamster, David G. Andersen, Hari Balakrishnan, and M. Frans Kaashoek. Measuring the effects of Internet path faults on reactive routing. In *Proceedings of the 2003 ACM SIGMETRICS International Conference on Measurement and Modeling of Computer Systems*, SIGMETRICS '03, pages 126–137, New York, NY, USA, 2003. ACM.

[109] B. Y. Zhao, Ling Huang, J. Stribling, S. C. Rhea, A. D. Joseph, and J. D. Kubiatowicz. Tapestry: A resilient global-scale overlay for service deployment. *IEEE J.Sel. A. Commun.*, 22(1):41–53, September 2006.

[110] Petar Maymounkov and David Mazières. Kademlia: A peer-to-peer information system based on the xor metric. In *Revised Papers from the First International Workshop on Peer-to-Peer Systems*, IPTPS '01, pages 53–65, London, UK, 2002. Springer-Verlag.

[111] Jian Gao and Cory Beard. Overlay networks to support internet emergency preparedness services. `http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.105.1410&rep=rep1&type=pdf`, January 2004. Accessed: 2016.

[112] Leslie Lamport. The part-time parliament. *ACM Trans. Comput. Syst.*, 16(2):133–169, May 1998.

[113] Peter Van Roy. Overcoming software fragility with interacting feedback loops and reversible phase transitions. In *Proceedings of International conference on Visions of Computer Science*, 2008.

[114] The thermodynamic system and its environment. The Columbia Electronic Encyclopedia. Columbia University Press, 6th edition, 2012.

[115] Farnoush Banaei-Kashani and Cyrus Shahabi. Criticality-based analysis and design of unstructured peer-to-peer networks as "complex systems". In *Proceedings of the 3st International Symposium on Cluster Computing and the Grid*, CCGRID '03, pages 351–, Washington, DC, USA, 2003. IEEE Computer Society.

[116] Ingo Scholtes, Jean Botev, Alexander Höhfeld, Hermann Schloss, and Markus Esch. Awareness-driven phase transitions in very large scale distributed systems. In *Self-Adaptive and Self-Organizing Systems, 2008. SASO '08. Second IEEE International Conference on*, pages 25–34, Oct 2008.

[117] A. Barrat and M. Weigt. On the properties of small-world networks. *The European Physical Journal B*, 13:547–560, 2000.

[118] C. P. Herrero. Ising model in small-world networks. *Phys. Rev. E*, 65(066110), 2002.

[119] H. Hong, B. J. Kim, and M. Y. Choi. Comment on "ising model on a small world network,". *Phys. Rev. E*, 66(018101), 2002.

[120] M. Kuperman and D. H. Zanette. Stochastic resonance in a model of opinion formation on small world networks. *The European Physical Journal B*, 26:387–391, 2002.

[121] A. Pękalski. Ising model on a small world network. *Phys. Rev. E*, 64(057104), 2001.

[122] J.-Y. Zhu and H. Zhu. Introducing small-world network effects to critical dynamics. *Phys. Rev. E*, 67, 2003.

[123] M. Takayasu, H. Takayasu, and K. Fukuda. Dynamic phase transition observed in the internet traffic flow. *Physica A*, 277:248, 2000.

[124] T. Ohira and R. Sawatari. Phase transitions in a computer network traffic model. *Phys. Rev. E*, 58(1):193–195, 1998.

[125] R. V. Solé and Sergi Valverde. Information transfer and phase transitions in a model of internet traffic. *Physica A*, 289:595–605, 2001.

[126] Erik Aurell and Sameh El-Ansary. A physics-style approach to scalability of distributed systems. In Corrado Priami and Paola Quaglia, editors, *Global Computing: IST/FET International Workshop, GC 2004 Rovereto, Italy, March 9-12, 2004 Revised Selected Papers*, pages 266–272. Springer Berlin Heidelberg, Berlin, Heidelberg, 2005.

[127] M. Mitchell. Self-awareness and control in decentralized systems. *Metacognition in Computation*, pages 80–85, 2005.

[128] P. R. Lewis, A. Chandra, S. Parsons, E. Robinson, K. Glette, R. Bahsoon, J. Torresen, and X. Yao. A survey of self-awareness and its application in computing systems. In *Self-Adaptive and Self-Organizing Systems Workshops (SASOW), 2011 Fifth IEEE Conference on*, pages 102–107, Oct 2011.

[129] Donatien Grolaux. Distributed graphical editor based on a transactional model (original title in french "editeur graphique réparti basé sur un modéle transactionnel"). Master's thesis, Université catholique de Louvain, Belgium, June 1998.

[130] Ruma R. Paul and Jérémie Melchior. Demo phases. `https://www.youtube.com/playlist?list=PLAebnT0VkXjFxkmph3mjkpJ-4fBQXN05`, 2016.

[131] Anant Agarwal, Jason Miller, Jonathan Eastep, David Wentziaff, and Harshad Kasture. Self-aware computing. Technical report, Massachusetts Institute of Technology, June 2009.

[132] Ananat Agarwal and Bill Harrod. Organic computing. Technical report, MIT CSAIL and DARPA IPTO, August 2006.

[133] H. Hoffman, M. Maggio, M. D. Santambrogio, A. Leva, and A. Agarwal. SEEC: A General and Extensible Framework for Self-Aware Computing. Technical report, MIT, November 2011.

[134] Peter Van Roy. Self management and the future of software design. *Electron. Notes Theor. Comput. Sci.*, 182:201–217, June 2007.

[135] Peter Van Roy, Seif Haridi, and Alexander Reinefeld. Designing robust and adaptive distributed systems with weakly interacting feedback structures. Technical report, ICTEAM, Université catholique de Louvain, Louvain-la-Neuve, Belgium, January 2011.

[136] L. Esterle, P. R. Lewis, M. Bogdanski, B. Rinner, and X. Yao. A socio-economic approach to online vision graph generation and handover in distributed smart camera networks. In *Distributed Smart Cameras (ICDSC), 2011 Fifth ACM/IEEE International Conference on*, pages 1–6, Aug 2011.

[137] J. Wang, D. Brady, K. Baclawski, M. Kokar, and L. Lechowicz. The use of ontologies for the self-awareness of the communication nodes. In *Proceedings of the Software Defined Radio Technical Conference (SDR'03)*, 2003.

[138] Mazeiar Salehie and Ladan Tahvildari. Self-adaptive software: Landscape and research challenges. *ACM Trans. Auton. Adapt. Syst.*, 4(2):14:1–14:42, May 2009.

[139] Pierre-Charles David and Thomas Ledoux. Towards a framework for self-adaptive component-based applications. In Jean-Bernard Stefani, Isabelle Demeure, and Daniel Hagimont, editors, *Distributed Applications and Interoperable Systems: 4th IFIP WG6.1 International Conference, DAIS 2003, Paris, France, November 17-21, 2003. Proceedings*, pages 1–14, Berlin, Heidelberg, 2003. Springer Berlin Heidelberg.

[140] W. Trumler, A. Pietzowski, B. Satzger, and T. Ungerer. Adaptive self-optimization in distributed dynamic environments. In *First International Conference on Self-Adaptive and Self-Organizing Systems (SASO 2007)*, pages 320–323, July 2007.

[141] Marc Shapiro, Nuno Preguiça, Carlos Baquero, and Marek Zawirski. A comprehensive study of convergent and commutative replicated data types. Technical report, Institut National de la Recherche en Informatique et Automatique (INRIA), Rocquencourt, France, January 2011.

[142] Basho Technologies. Riak distributed database. `http://basho.com/riak/`, 2015. Accessed: 2016.

[143] Giuseppe DeCandia, Deniz Hastorun, Madan Jampani, Gunavardhan Kakulapati, Avinash Lakshman, Alex Pilchin, Swaminathan Sivasubramanian, Peter Vosshall, and Werner Vogels. Dynamo: Amazon's highly available key-value store. *SIGOPS Oper. Syst. Rev.*, 41(6):205–220, October 2007.

[144] Seth Gilbert and Nancy Lynch. Brewer's conjecture and the feasibility of consistent, available, partition-tolerant web services. *SIGACT News*, 33(2):51–59, June 2002.

[145] Karin Petersen, Mike J. Spreitzer, Douglas B. Terry, Marvin M. Theimer, and Alan J. Demers. Flexible update propagation for weakly consistent replication. *SIGOPS Oper. Syst. Rev.*, 31(5):288–301, October 1997.

[146] SyncFree Consortium. Protocols for crdts in small-scale full replication. `https://syncfree.lip6.fr/attachments/article/46/WP2-report.pdf`, 2014. Accessed: 2016.

[147] Mustaque Ahamad, Gil Neiger, James E. Burns, Prince Kohli, and Phillip W. Hutto. Causal memory: definitions, implementation, and programming. *Distributed Computing*, 9(1):37–49, 1995.

[148] Wyatt Lloyd, Michael J. Freedman, Michael Kaminsky, and David G. Andersen. Don't settle for eventual: Scalable causal consistency for wide-area storage with cops. In *Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles*, SOSP '11, pages 401–416, New York, NY, USA, 2011. ACM.

[149] H. Attiya, F. Ellen, and A. Morrison. Limitations of highly-available eventually-consistent data stores. *IEEE Transactions on Parallel and Distributed Systems*, PP(99):1–1, 2016.

[150] Basho Technologies. basho_bench. `https://github.com/basho/basho_bench`, 2016. Accessed: 2016.

[151] Márk Jelasity, Spyros Voulgaris, Rachid Guerraoui, Anne-Marie Kermarrec, and Maarten van Steen. Gossip-based peer sampling. *ACM Trans. Comput. Syst.*, 25(3), August 2007.

[152] Márk Jelasity, Rachid Guerraoui, Anne-Marie Kermarrec, and Maarten van Steen. The peer sampling service: Experimental evaluation of unstructured gossip-based implementations. In *Proceedings of the 5th ACM/IFIP/USENIX International Conference on Middleware*, Middleware '04, pages 79–98, New York, NY, USA, 2004. Springer-Verlag New York, Inc.

[153] Roberto Baldoni, Marco Platania, Leonardo Querzoni, and Sirio Scipioni. Practical uniform peer sampling under churn. In *Proceedings of the 2010 Ninth International Symposium on Parallel and Distributed Computing*, ISPDC '10, pages 93–100, Washington, DC, USA, 2010. IEEE Computer Society.

[154] Y. Huang, C. Kintala, N. Kolettis, and N. D. Fulton. Software rejuvenation: analysis, module and applications. In *Fault-Tolerant Computing, 1995. FTCS-25. Digest of Papers., Twenty-Fifth International Symposium on*, pages 381–390, June 1995.

[155] K. Vaidyanathan and K. S. Trivedi. A comprehensive model for software rejuvenation. *IEEE Transactions on Dependable and Secure Computing*, 2(2):124–137, April 2005.

[156] Yves Jaradin and Peter Van Roy. A formal model of software rejuvenation for long-lived large-scale applications, March 2011. Draft Report.

[157] Planetlab. `https://www.planet-lab.org/`, 2007. Accessed: 2016.