ROYAL INSTITUTE
OF TECHNOLOGY

On the Performance Analysis of Large Scale, Dynamic, Distributed and Parallel Systems.

JOHN ARDELIUS

Doctoral Thesis
Stockholm, Sweden 2013

TRITA-ICT/ECS AVH 13:20
ISSN 1653-6363
ISRN KTH/ICT/ECS/AVH-13/20-SE
ISBN 978-91-7501-918-5

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

Akademisk avhandling som med tillstånd av Kungl Tekniska högskolan framlägges till offentlig granskning för avläggande av teknologie doktorsexamen i datalogi Tisdag den 10 December 2013 klockan 10:00 i sal 431 i Forum IT-Universitetet, Kungl Tekniskahögskolan, Isajordsgatan 39, Kista.

Swedish Institute of Computer Science
SICS Dissertation Series 67
ISRN KTH/ICT/ECS/AVH-13/20-SE
ISSN 1653-6363.

Tryck: Universitetsservice US AB

Abstract

Evaluating the performance of large distributed applications is an important and non-trivial task. With the onset of Internet wide applications there is an increasing need to quantify reliability, dependability and performance of these systems, both as a guide in system design as well as a means to understand the fundamental properties of large-scale distributed systems. Previous research has mainly focused on either formalised models where system properties can be deduced and verified using rigorous mathematics or on measurements and experiments on deployed applications. Our aim in this thesis is to study models on an abstraction level lying between the two ends of this spectrum. We adopt a model of distributed systems inspired by methods used in the study of large scale system of particles in physics and model the application nodes as a set of interacting particles each with an internal state whose actions are specified by the application program. We apply our modeling and performance evaluation methodology to four different distributed and parallel systems.

The first system is the distributed hash table (DHT) Chord running in a dynamic environment. We study the system under two scenarios. First we study how performance (in terms of lookup latency) is affected on a network with finite communication latency. We show that an average delay in conjunction with other parameters describing changes in the network (such as timescales for network repair and join and leave processes) induces fundamentally different system performance. We also verify our analytical predictions via simulations. In the second scenario we introduce network address translators (NATs) to the network model. This makes the overlay topology non-transitive and we explore the implications of this fact to various performance metrics such as lookup latency, consistency and load balance. The latter analysis is mainly simulation based. Even though these two studies focus on a specific DHT, many of our results can easily be translated to other similar ring-based DHTs with long-range links, and the same methodology can be applied even to DHT's based on other geometries.

The second type of system studied is an unstructured gossip protocol running a distributed version of the famous Belman-Ford algorithm. The algorithm, called GAP, generates a spanning tree over the participating nodes and the question we set out to study is how reliable this structure is (in terms of generating accurate aggregate values at the root) in the presence of node churn. All our analytical results are also verified using simulations.

The third system studied is a content distribution network (CDN) of interconnected caches in an aggregation access network. In this model, content which sits at the leaves of the cache hierarchy tree, is requested by end users. Requests can then either be served by the first cache level or sent further up the tree. We study the performance of the whole system under two cache eviction policies namely LRU and LFU. We compare our analytical results with traces from related caching systems.

The last system is a work stealing heuristic for task distribution in the TileraPro64 chip. This system has access to a shared memory and is therefore classified as a parallel system. We create a model for the dynamic generation of tasks as well as how they are executed and distributed among the participating nodes. We study how the heuristic scales when the number of nodes exceeds the number of processors on the chip as well as how different work stealing policies compare with each other. The work on this model is mainly simulation-based.

Sammanfattning

Att utvärdera prestanda hos storskaliga distribuerade system är en viktig och icke-trivial uppgift. I och med utvecklingen av Internet och det faktum att applikationer och system har fått global utsträckning, har det uppkommit ett ökande behov av kvantifiering av tillförlitlighet och prestanda hos dessa system. Både som underlag för systemdesign men också för att skapa förståelse och kunskap om fundamentala egenskaper hos distribuerade system.
Tidigare forskning har i mångt och mycket fokuserat antingen på formaliserade modeller, där egenskaper kan härledas med hjälp av strikta matematiska metoder eller på mätningar av riktiga system. Målet med arbetet i denna avhandling är att undersöka modeller på en abstraktionsnivå mellan dessa två ytterligheter. Vi tillämpar en modell av distributerade system med inspiration från så kallade partikelmodeller från den teoretiska fysiken och modellererar applikationsnoder som en samling interagerande pariklar var och en med sitt eget interna tillstånd vars beteende beskrivs av det exekvernade programmet i fråga. Vi tillämpar denna modelerings- och utvärderingsmetod på fyra olika distribuerade och parallella system.
Det första systemet är den distribuerade hash tabellen (DHT) Chord i en dynamisk miljö. Vi har valt att studera systemet under två scenarion. Först utvärderar vi hur systemet beteer sig (med avseende på lookup latency) i ett nätverk med finita kommunikationsfördröjningar. Vårt arbete visar att en generell fördröjning i nätet tillsammans med andra parametrar (som t.ex. tidsskala för felkorrektion och anslutningsprocess för noder) genererar fundamentalt skilda prestandamått. Vi verifierar vår analytiska model med simuleringar. I det andra scenariot undersöker vi betydelsen av NATs (network adress translators) i nätverksmodellen. Förekomsten av dessa tar bort den transitiva egenskapen hos nätverkstopologin och vi undersöker hur detta påverkar lookup-kostnad, datakonsistens och lastbalans. Denna analys är främst simuleringsbaserad. Även om dessa två studier fokuserar på en specifik DHT så kan de flesta resultat och metoden som sådan överföras på andra liknande ringbaserade DHTer med långa länkar och även andra geometrier.
Den andra klassen av system som analyseras är ostrukturerade gossip protokoll i form av den välkända Belman-Ford algoritmen. Algoritmen, GAP, skapar ett spännande träd över systemets noder. Problemställningen vi studerar är hur tillförlitlig denna struktur, med avseende på precisionen på aggregat vid rotnoden, är i ett dynamiskt nätverk. Samtliga analytiska resultat verifieras i simulator.
Det tredje systemet vi undersöker är ett CDN (content distribution system) med en hierarkisk cache struktur i sitt distributionsnät. I den här modellen efterfrågas data från löven på cache-trädet. Antingen kan förfrågan servas av cacharna på de lägre nivåerna eller så skickas förfrågan vidare uppåt i trädet. Vi analyserar två fundamentala heuristiker, LRU och LFU. Vi jämför våra analytiska resultat med tracedata från riktiga cachesystem.
Till sist analyserar vi en heuristik för last distribution i TileraPro64 arkitekturen. Systemet har ett centralt delat minne och är därför att betrakta som parallellt. Vi skapar här en model för den dynamiska genereringen av last samt hur denna distribueras till de olika noderna på chipet. Vi studerar hur

heuristiken skalar när antalet noder överstiger antalet på chipet (64) samt jämför prestanda hos olika heuristiker. Analysen är simuleringsbaserad.

Till Regina, Hugo och Frank

Acknowledgements

Many people have contributed to this thesis in one way or the other and I would like to take this opportunity to acknowledge some of them. First and foremost I would like to thank my supervisor Supriya Krishnamurthy for patiently overseeing and supporting the progress and development of my research abilities during the past years. Through many frutiful discussions I have learned a lot about probabilistic modeling of dynamic systems in an interesting problem domain. I would also like to thank my supervisor Seif Haridi for his support, great enthusiam, interesting discussions and introduction to the distributed systems community.

Further I would like to thank all my co-authors and collaborators for greatly contributing to this thesis. Doing research in an intersection of many diciplines requires a broad domain knowledge, something you have greatly contributed with. In the process I have had the opportunity to learn about a variety of intersting problem domains.

I would also like to thank the members of the CSL lab and other co-workers at SICS for many interesting discussions and ideas. A special thanks to Niklas Ekström for reading through the thesis draft and his many thoughtful remarks.

Last but not least I would like to thank my high school teacher Charlotte Nilsson and to some extent also Andreas Ricci who sparked my interest in mathematics, something that led me to KTH in the first place.

Contents

Chapter 1

Introduction

Distributed computing is the part of computer science concerned with issues regarding how systems consisting of several concurrently executing computation devices behave and function. In its most general form a distributed computing system is a set of computation devices, *nodes*, connected through a communication network of some form. The behaviour of a distributed system differs from the behaviour of a single computation device in that each node only has access to local, node specific, state information and that execution of tasks are done in parallel rather then in series.

Even though most systems can fit into this abstraction (biological systems of communicating cells, sociological systems of communicating people, physical systems of interacting particles) a general implicit assumption in the study of distributed systems is that the nodes are computing devices (computers) connected via some communication channel. Each computer can execute programs that perform operations on its local data and communicate with other computers to which it is connected. The main question in the study of distributed systems is then what such a collection of computers can do together which a single computer cannot.

A further distinction is often made between a *parallel* system and a *distributed* system in that in the former each node has access to a global shared memory where no such thing exists per se in the latter. A characterising aspect of all the systems studied here, regardless of the presence of a shared memory, is that each node executes its program *independent* of the execution of the other nodes. This implies that if a node fails or becomes corrupt in some way, the other nodes will not be automatically informed about this event. This implicit strong dependence on the notion of *time* is perhaps what differentiates distributed computing from single node computing. Different assumptions about how fast messages, and thereby information, can travel in the system have given rise to a further distinction between *synchronous* systems where there is an upper limit to how long a request/response will take and *asynchronous* system which lacks such a limit [42].

The research field of distributed systems has a strong connection to engineering and system construction and many of the motivating problems and their settings are defined in close connection with current industrial trends. Much attention has been given to the question of how to best *design* a system to solve a particular problem in areas ranging from parallel computing systems, operating systems, wireless sensor networks, Internet applications, telecommunication systems and more. In building these systems general properties and associated issues have been found that distinguishes this field of study from other computer science fields namely, the notion of timing, concurrency, fault tolerance and synchronization.

In this thesis we are concerned with the *performance* of distributed systems, that is, how well does a particular system match its intended design criterion and how does the operational outcome of a system compare to that of other similar ones? We have focused on large scale, loosely coupled systems in that the number of participation nodes is possibly very large and the dependence between the states and actions among the nodes is relatively small. More generally, the common denominator of the systems studied is that they are all highly *dynamic*, *large scale* and contain some inherently *stochastic* properties. This model applies, as our work shows, to a wide range of systems from different problem domains.
An inherent difficulty in analysing a distributed system is that the number of possible states the system can be in grows very quickly with number of participating nodes. Things get even more complicated when the state evolution of the system is governed by stochastic rather than deterministic processes. In this thesis we apply a set of probabilistic tools traditionally used in the study of dynamic systems in physics and chemistry to a variety of distributed computing systems. We show that by using a set of modeling and analysis techniques we can accurately predict system performance of many highly dynamic, large scale systems for a wide range of parameters. Much of the work done in this thesis have been conducted at the Swedish Institute of Computer Science (SICS) in close connection with both industrial and academic partners all confirming the need for models and analytical tools to better understand the properties of large complex systems.

1.1 Summary of research objectives

The aim of the work done in this thesis has been to both establish a methodology for how to model and evaluate the performance of large scale dynamic systems as well as applying this method to a set of important distributed systems and applications. With the onset of large scale peer-to-peer applications, it has become important to include *churn* (users constantly joining and leaving a system) in system models. Many other systems have the same property namely that they are designed to be highly scalable in the presence of various stochastic input. Currently there is no straight forward way to model and analyse these systems. Many analytical work of performance analysis are done on a very conceptual system level or/and require very specific mathematical tools. Our aim has been to establish a methodology which can be applied by a broader computer science (and engineering) community to analyse distributed applications. For the individual systems the performance metrics differ but a common denominator is *scalability*, perhaps the most important property of systems in the Internet era.

1.2 Main contributions

Apart from establishing a methodology for performance analysis the main contributions of the thesis include various system specific results. These results will be presented in more detail in the coming chapters but are stated here in summary. They include:

- A DHT model including explicit network delays. Using this model we study the interplay between lookup latency and correctness of routing table entries. The model predicts that lookup latencies will scale faster than the commonly accepted $O(log(N))$ even for low delays.

- An evaluation of the impact of network address translators to DHT performance. Previous research have mainly assumed the underlying network to be a complete graph. When this is not true various problems occur such as heterogenous load, lookup inconsistencies, long entry time for new nodes and more. This is the first work studying these issues.

- An analytic model of an unstructured overlay running a spanning tree-heuristic. Unstructured overlays have many applications and the famous Belman-Ford protocol is widely used. Despite its popularity there are few (if any) analytical works to evaluate system performance. We construct a model that is able to predict the performance very accurately when compared with simulations. The work also states an interesting balance conjecture regarding the transitions between states in the spanning tree which seem to hold for a wide range of parameters.

- A fluid model for an in-network cache hierarchy together with asymptotic results for the least recently used (LRU) cache eviction policy. Using the model we are able to predict, among other things, the impact of adding an additional layer of caches, the impact of larger cache sizes and more.

- A stochastic model for task generation in a general multi-core system. Using this model we are able to quantify the performance of various work stealing heuristics in the execution of benchmark algorithms. Such a modeling approach is novel and enables us to study scalability issues for a large number of computing cores.

1.3 Thesis organisation

The thesis is organised as follows. First we present the systems studied in the thesis, where and how they are used and how we model them. This is followd by a set of scientific questions we intend to answer and the set of performance metrics we examine for each system. The main part of the introduction is contained in the methodology section where we describe how to approach our scientific questions. We describe modeling techniques as well as the analytical and numerical methods used in the thesis. Some alternative tools and limitations are discussed. The final chapter of the introduction contains a summary of the most relevant findings and results for each system studied followed by five chapters with the scientific papers describing the details of the work.

Chapter 2

Choice of systems studied

2.1 General concepts

Common to all distributed systems is that they can be modeled using the same basic components namely

- a set of nodes N
- each with a set of internal states $\bar{x}$
- connected through a network graph $G(V, E)$

All of these can in general be thought of as dependent on a global measure of time t.

The focus of the work done in this thesis is to study systems where some components are *non-deterministic*. It could be either that the set of nodes N changes with time in some way, the links in the underlying graph $G(E, V)$ change or the local state of a node changes in a random fashion. We model these non-deterministic properties as stochastic processes that require us to analyse the model from a probabilistic point of view.

This chapter contains a brief overview of the systems studied and how they are modeled in terms of the above parameters. We also discuss the basis for stochasticity in these systems and how they are modeled in each case. The chapter is intended as a brief overview of the types of systems studied. Details of the systems will be presented together with the analysis in later chapters. We start out by a general discussion about overlay networks and churn common to all systems studied in this thesis.

Overlay networks

An important class of computer networks is so called *overlay* networks. An overlay network is a virtual network created on top of an existing network. In many practical cases, the network topology we are studying is not a direct mapping of the lower layers of the network communication stack such as the data link or physical layers. Rather, we are interested in how nodes, operating in the application layer, interact and behave.

If we study an Internet application for instance, we are generally not interested in the properties of the physical network with routers and hubs but rather the network consisting of end user computers. This network is then an overlay on top of the Internet. An implicit assumption in the case of Internet applications is that the nodes can in principle communicate with all other nodes over the physical network but are not able to do so due to large overhead related to maintaining many live communication links. In other applications, such as a wireless network, each node has restricted communication range and the overlay more closely resembles the physical environment in which the system is deployed. These restrictions and differences are incorporated in the model of the connection graph $G(E, V)$.

A further distinction between different types of overlay networks is made between *structured* and *unstructured* overlay networks. In an *unstructured*, or ad-hoc system, the nodes do not follow any particular rule regarding which other nodes to connect and communicate with. An unstructured system also implies a high level of symmetry in the overlay topology. Since there is no a-priori rule for how the nodes should connect, an application running on top of the network should be able to function even if the nodes are permuted arbitrarily. Many computational problems (such as counting the number of nodes in a system, averaging a set of local values etc) can be solved efficiently without an overall structure in the underlying connection graph $G(E, V)$. Unstructured networks are found in many applications ranging from wireless sensor networks to large scale Internet applications.

In a *structured* network on the other hand, each node knows, according to the application protocol which other (virtual) nodes to connect to. This rule induces a particular structure in the connection graph and this asymmetry can be exploited by applications to perform more advanced tasks then possible for fully homogenous systems.

The systems studied in this thesis are all overlay networks. Some are structured and some are unstructured. Some systems run on top of the Internet and some run on a processing chip. The common demoninator is that they all have dynamic aspects which can be modeled as stochastic and we are interested in how system parameters scale with these processes.

Traffic models

We are interested in analysing the performance of systems in a deployed, live setting. For systems where the number of participating nodes varies with time, such as in an peer-to-peer application, an important part of our model is the manner in which this quantity varies. The process of nodes constantly joining and leaving the system is called *churn*. From the systems perspective, churn is in a sense a

non-deterministic process since there is no a priori information about when a node will join or leave the system. This requires us to consider a probabilistic description of these systems and is the basis for stochasticity in the models studied. There is a trade-off between being able to solve a highly detailed model and keeping enough information for the model to be useful. There are a number of ways in the literature to model join and leave processes referred to as *traffic models* [6].
The most basic and generic model for churn is to assume that all join and leave events are uncorrelated and the time spent in the system is exponentially distributed. These assumptions allow us to model churn as a set of Poisson process for the join and leave events.
An alternative is to study more generic distributions for the time a node spends in the system. Such processes are called renewal processes and have been used to model node churn in peer-to-peer networks [123]. These models still assume that the join and leave processes of different nodes are independent of each other and have well defined time independent distributions, but do not necessarily assume these distributions to be Poisson.
Fluid models, which are described in more detail in the methodology chapter, are also common ways to model network traffic.
In this thesis we use the Poisson churn model. The Poisson model has a number of benefits. First, since it is memoryless it fits easily in our Markovian model description. It has also the property that any combination of Poisson processes is itself Poisson. Apart from being simple to analyse the Poisson process has a very appealing property expressed through the Palm-Khintchine theorem namely that a large enough set of non-Poisson processes can with high accuracy jointly be well described by a Poisson process. [92, 123]

2.2 Chord - a distributed hash table

A distributed hash table (DHT) is a generic system that performs a key-value lookup service over a set of participating nodes. The system is able to store data items (values) and access them using short hand labels (keys). It performs the same operation as the hash table data structure with the benefit of scalability. The available storage space for the DHT scales linearly with the number of participating nodes (if data is stored without replication) so a DHT is well suited for storage of very large data sets such as web indices, global file storage and more. A DHT is generally deployed in Internet wide applications.

A DHT is an example of a *structured* overlay topology in that each node is designated to connect to a specific set of other nodes.

The set of nodes $N(t)$ in a DHT are the nodes connected to the system at time t and the graph $G(E, V, t)$ is all the connections (uni- or bidirectional) held by the nodes at this time. Each node has its own local state about which set of keys it is responsible for (meaning it will store the data stored under that key) as well as when it will do maintenance. We are interested in doing an analysis of the average

time it takes for Chord to perform a lookup, that is the system *latency*. In order to do this our node models need to contain state information about whether or not the entries in the nodes routing table are correct or not.
As mentioned above, we use a Poisson traffic model to describe the $N(t)$ process.

2.3 Work stealing on the Tilera chip

A multi core processor is a processing unit consisting of several CPU:s. Due to the fast increase in capacity of modern CPU:s, one current trend in the design of processing chips is to interconnect a large set of smaller computational units. This has several advantages such as reduced heat generation as well as increased scalability. Drawbacks are however that the CPU:s (cores) need to communicate and synchronize their state in order for the whole system to perform in a predictable manner. Traditional (single core) programming languages do not easily translate to a multi core environment. Due to state dependence between various parts of a program it is sometimes hard to exploit the maximum benefit (linear performance increase) from a multi core system. One approach is to design new programming languages and procedures so that the nodes can perform as much work as possible in parallel. This implies communicating with each other about how much work each node has to do and if some of it is possible to hand out to a node with available capacity. A multi core system is then possible to model as a general distributed system. The set N are the cores and $G(E, V)$ the (often fully connected) communication paths and the internal state contain the current execution of the program at a given node. The stochastic part of the model is how work load is generated in the executing nodes. In the execution of a program, each computational task can with some probability spawn a new set of computation tasks. These tasks can then be assigned to other nodes in an operation known as "stealing". In our model, the spawning of new tasks is modeled as a stochastic process where each task requires an exponentially distributed time to finish. When done, each task spawns k new tasks with some fixed probability α.

2.4 In-network, hierarchical caching

Currently, most of the data downloaded over the Internet is video [26]. Traffic related to video on demand services such as YouTube and Daily motion are responsible for a large percentage of the transferred volume as per mid 2012. From the network operators perspective, the increase in traffic volume could potentially lead to network congestion, increased response times and in the end, unhappy customers. However, many requests for video data are redundant in the sense that people tend to watch the same movies. There is then a strong incentive to *cache* popular items close to the end-user in the network to reduce overhead transfer. Many network providers as well as a lot of current research [7] focus on how to best construct a system of caches which is economically feasible and still decrease overall

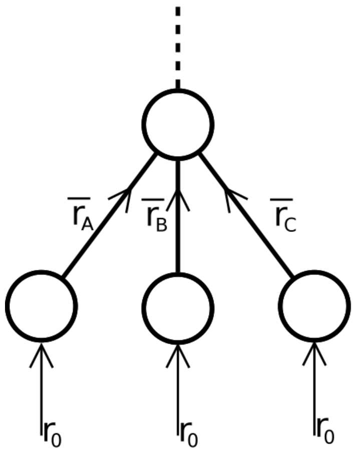


Figure 2.1: End-user requests reach the first level of caches. The traffic not served is relayed upwards the cache hierarchy.

network load. One such approach is to use so called *in network caches*. These are caches that are put within the existing distribution network at strategic places. The network closest to the end-user is called an access aggregation network which can be thought of as a hierarchical network topology with the top level connecting the network to the Internet. In our model framework the connection graph $G(E, V)$ can be modeled as a tree with end-user requests sent to the caches at the leaves and the Internet connection at the root. The nodes N are the caches and the internal state $\bar{x}$ is the content held in memory. Requests from end-users reach the first level of caches which can either serve the request if the content is present in the cache or relay it further in the cache hierarchy. We model the arrival of end-user requests as Poisson processes with parameters λ_n for content n. To incorporate the heterogenous popularity among content the process rates are drawn from a Pareto distribution with parameter α.

2.5 Distributed Belman-Ford - a gossip algorithm

Asynchronous protocols where the nodes exchange pairwise pieces of information to solve a global problem are refered to as *gossip algorithms*. Although not a system in itself, gossip algorithms are used as building blocks in many systems to provide data dissemination and replication, aggregate information and more [107]. A famous and well used algorithm for solving the single-source shortest path problem in a distributed way is the Bellman-Ford algorithm. It does so by iteratively updating the distance for each node to the source node, or the *root*.
A classical application of Bellman-Ford is distance-vector routing. From a per-

formance perspective we would like to understand how an implementation of the distributed Bellman-Ford algorithm, known as GAP, behaves in the presence of node churn. We model a set of nodes N, connected randomly to k other nodes in a graph $G(E, V)$. k is drawn from an Poisson distribution. The local state $\bar{x}$ is the local information about its perceived distance from the root. At each update round, the nodes probe their neigbours and update their local state to the lowest distance of their neigbours plus one. One of the neigbours with the lowers level is selected as a parent.
We are interested in analysing the performance of GAP and the metric we choose is how well the root node estimates the total number of participating nodes. Each node independently issues a call its neigbours and sums up the replies recieved from its offspring nodes, adding itself to the count. The value of the sum at the root is the an estimate of the true number of participating nodes.

To analyse the correctness of this estimate we contruct two Markov processes. One that keeps track of how many nodes in the system currently have level x and another that keeps track of the average aggregate count held by a node at level x. The estimated value at the root (level 0) is then compared to system simulations.

Chapter 3

Scientific questions

3.1 Performance analysis of dynamic systems

The general question addressed in this thesis is how we can evaluate the performance of dynamic, distributed systems. As mentioned in the previous chapters, distributed systems are hard to analyse in full due to the large number of states we need to consider. A useful performance analysis then needs to include both a *modeling* step where we try to capture the most important properties of the system as well as an analytic treatment, that is, *solving* the model. The aim of this thesis is to provide a set of probabilistic tools, presented in the coming chapter, which can be used to model and analyse the evolution of distributed systems. Using these tools we can formulate and answer questions like- How do certain patterns for user joins and leaves affect lookup performance of a DHT? How does a particular cache eviction policy affect the throughput of an access network? How is the run-time of a certain algorithm affected by the way a work stealing algorithm samples work load? What is the tradeoff between communication rate and accuracy of an unstructured counting algorithm? Further we would like to answer these questions when the number of participating nodes in the system grows large.
What these questions have in common is that they all ask how a design choice in the algorithm running on a local node, affects the system as a whole. How does the design of the microenvironment translate to the macro environment?

In this thesis the methodological tools described in the coming chapter are applied to four distributed systems, a distributed hash table (DHT), an unstructured gossip algorithm, a work stealing protocol in a multicore environment and a content distribution network (CDN). Common to these systems is that they are all affected by dynamic, essentially stochastic input, all need to scale to a large number of nodes and all need to perform well under various conditions. Further, in practice, the designer of these systems will have to set values for a number of system parameters such as cache size, update frequencies, number of routing table entries etc. without being hundred percent certain about how these will affect the overall system performance. The aim of a performance analysis is hence also to guide designers on how endogenous parameters should be set and how they affect the system.

This chapter will describe, in more detail, the set of questions we have analysed for the individual systems and how they relate to other works on similar subjects.

3.2 Scalability - size and time

As mentioned in the introduction *scalability* is one of the most important properties of a distributed application. Most of the time we want to understand how a particular system will behave when the number of participating nodes grows large. Another physical parameter of interest is *time*. Behaviour at start-up will most probably differ from how the system behaves under steady conditions. In this thesis we are mainly interested in how a system behaves in steady state. Steady state is important since the system will most probably reside in or near this state for a very long time while it is in operation. The steady state has a very convenient analytical property namely that it is time independent. Our modeling methodology allows us however to study other transient properties as well.
The benefit of using an explicit CTMC description of a system is that we can study explicit scaling relations rather than rely on asymptotic results when $N \to \infty$. The main question we are interested in is then "How do the system properties in steady-state scale with the number of nodes?".
This issue is the common denominator in all our analysis and is complemented by system specific questions stated below.

3.3 System specific performance metrics

Lookup latency in Chord

The peer-to-peer computer architecture has attained a great deal of attention in the research community over the past decade [4]. With the onset of large-scale file sharing applications such as Napster and Gnutella [85], researchers began to investigate what kind of applications could be created in such a fully distributed fashion. A key property of a peer-to-peer system is that all the participating nodes have *equal responsibility.* That is, the system is fully symmetrical in terms of interchanging one node for another. The nodes run the same algorithm and are in many cases considered to have the same physical properties such as available bandwidth and storage.

Another important property of a peer-to-peer system is that due to its inherent lack of central maintenance, nodes have to be able to join and leave the system using procedures including only (not too many) other peers. Such a dynamic process of node join and leave is known as *node churn*. Further, the nodes need to maintain the system by performing local periodic operations.

The study of a peer-to-peer system is therefore the study of (an a priori) fully symmetrical dynamic distributed systems.

In practice, one very fundamental service to provide for a distributed data sharing application is a key-value store. A key-value store is a data structure that can perform storage and retrieval of data. When data is stored, a key is generated which later can be used to retrieve the data. This abstraction can be implemented using a hash table. Many attempts have been made to implement a distributed hash table (DHT) [85] which is able to store and address large amounts of data even under churn, one of the most well studied being the Chord system [109]. The Chord system claims to be able to retrieve data, or perform lookups, which is the number of network hops required to find a piece of data , in $O(log(N))$ number of network hops, N being the number of participating nodes in the system. Many similar systems have been created by different research groups [85] with the same performance complexity. This would imply that it is possible to store and fetch substantial amounts of data under dynamic conditions. DHTs hence appear to be a promising building block for large-scale peer-to-peer applications.

One very crucial assumption in many of the analytical works in peer-to-peer systems is that of a perfect network (this abstraction is used, often implicitly); it is assumed that the underlying connectivity graph is well approximated by a full mesh and there exists a communication path between each pair of nodes. Further, messages sent over the network are assumed (often again implicitly) to be delivered instantaneously or at least on a much faster time scale than other relevant processes (such as join and fail). Here we refer to this model as the perfect network model.

In this thesis we have investigated how Chord (and more generally all similar DHT:s) are affected when we deviate from the perfect network assumption. What happens when network delays are of the same order of magnitude as joins and leaves? How is the performance affected if the underlying graph is *not* a full mesh. In practice, solving network issues are a big part of designing a distributed application but most theoretical work do not consider these problems. Our aim here is to bridge this gap.

Accuracy of a generic aggregation protocol - GAP

As mentioned in the introduction, peer-to-peer systems can be either structured, as in the case of Chord, or unstructured. One appealing property of an unstructured system is that it requires even less coordination among the nodes. So besides being fully symmetrical on the node level, an unstructured system displays a high degree of symmetry on the overlay level. The resulting overlay network created by the application does not provide any unique identifiers to the participating nodes and hence does not easily allow for point-to-point routing. However, many important

distributed computation problems as well as network monitoring can be efficiently solved without unique identifiers.
An important class of unstructured networks is those running *gossip* algorithms. In these networks data is disseminated by nodes exchanging messages among their nearest neighbours. A set of messages is received from a nodes neighbours, based on which it calculates an outgoing message which it passes on to a subset of its neighbours depending on policy. Such a simple protocol is, due to its simplicity, very robust but can still perform very useful operations [107].
How to model and evaluate the performance of unstructured gossip based algorithms is still an open question [21] and has been addressed in this thesis using the methodology described in the next chapter. We have selected a version of the simple but powerful Belman-Ford algorithm [14]. This counts the number of participating nodes in the system by constructing a spanning tree on top of an underlying communication graph. A special node, the *root*, provides an estimate of the number of nodes in the network. The accuracy of this estimate depends on the number of nodes in the network, the frequency of gossiping messages as well as the number of entries in the routing tables (neighbours) in a non trivial way.
The issue we address is how one can evaluate the performance of such a highly dynamic and distributed system.

Performance of in-network caching hierarchies in content distribution networks (CDN)

If a gossip based ad-hoc network is one of the more unstructured networks, a static network of servers is perhaps one of the more structured. Many ISP's maintain large content delivery networks that connect end-users to the back-bone Internet. These networks have limited capacity in terms of bandwidth. One common way for the ISP to reduce the amount of traffic running in the network and hence increase the amount of available bandwidth (in order to postpone major upgrades of the physical network infrastructure) has been to introduce caches inside the network. The caches store popular items close to the end-users so that it can be received without using the scarce resources in the core network.
The question now is which policy these caches should operate under? Since the caches have limited storage capacity it is not possible to store all passing data items. How can we create a model of a such a complex caching network with a multitude of objects being requested by a large number of users?

Performance of many core work stealing heuristics

One challenging task for the computer architecture community is to keep increasing the computational power of central processing units (CPU). The famous Moore's law , which states that the maximum number of transistors that fit on an integrated circuit grows exponentially with time, has been seen to hold for almost three decades. Eventually however, many researchers claim, in order to keep in-

creasing computational capacity the processors will have to work in parallel rather than in series, in a multi-core architecture.
How to make programs run on a multi-core platform is a topic of much recent research. One approach is to design new programming languages and coding conventions, which are better adapted to a parallel environment. One central issue here is how the nodes should share and coordinate work load in an efficient way, given a particular algorithm.
We have created a stochastic model for the generation and distribution of work load among N interconnected processors to examine how a so called work stealing architecture [16] performs on a TileraPro64 architecture. The chip has 64 cores and the execution of an algorithm begins with one core starting to execute operations. When the program reaches a point which allows for parallel execution, it puts the new execution path (a.k.a. a task) in a local pool. Other cores are then free to 'steal' this task and when done, return the output to the owner. The issue we are addressing is how to best evaluate a work stealing procedure given the wide range of possible algorithms that can be executed on a multicore chip?

Chapter 4

Scientific method

4.1 Modeling a distributed system

Distributed systems research is conducted both from an applied, systems oriented, point of view as well as plain theoretical. The applied research community is interested in how to construct and deploy (often large scale) distributed applications, how to measure their performance in a real setting and hopefully be able to extract some general principles which can be used as a guide to enhance the design of future systems. The theoretical research community on the other hand has been dealing with fundamental, logical, questions of distributed computing such as when and if a certain distributed algorithm will terminate, if outputs from different parts of the system will be consistent under certain conditions and under which conditions it is possible to reach consensus among the participating nodes. The difference between the two approaches is that the applied research (apparent from the term applied) deals with the physical world of network delays, user patterns, firewalls, coding issues, etc etc.. whereas the theoretical approach inevitably needs to rely on an abstraction, or *model* of the physical world. An integral part of any theoretical work is then not only to solve but to construct models.
It is important to keep in mind, from a conceptual point of view, that models, not only of distributed systems, but in general, all have a *purpose*. A model is created to explain or compare something, to argue for a certain action or for other similar reasons. The important difference between studying the real world (which has no default purpose) is that the model cannot be fully separated from the context in which it is created. When working with models we then need to specify both its context and purpose in order for it to be meaningful. Since the topic for this thesis is how to analyse models of distributed systems, we should hence start by clarifying the context in which models fit in and can serve their purpose.

In [19] four different purposes are stated for models of computer systems of the type studied in this thesis. They all try to explain system properties which are listed below together with their definitions from the "IEEE 90 IEEE Standard Glossary of Software Engineering Terminology".

1. *Performance.* The degree to which a system or component accomplishes its designated functions within given constraints, such as speed, accuracy, or memory usage.

2. *Reliability.* The probability that the software will not cause the failure of the system for a specified time under specified conditions.

3. *Availability.* The ability of a system to perform its required function at a stated instant or over a stated period of time.

4. *Dependability.* The ability to deliver a service that can justifiably be trusted.

We see that in order to study performance or availability we need to know the designated function of the systems as well as its constraints. To study reliability or dependability we need to specify what we mean by system failure and expected function. All these properties are often implicit in the study of distributed systems but should be kept in mind when we continue and construct our models. In this thesis we are mainly concerned with the first item namely, *performance analysis*.

For the purpose of performance analysis, a model of a distributed system should contain, apart from an abstraction of the physical environment, an abstraction of the actual algorithm running on the participating nodes. If we use the general finite-state automaton model [116] for the participating computation nodes, the number of available states when we consider the entire code running on an application machine (operating system, network stack, other running applications etc) becomes too large to handle. If we narrow our analysis down and only look at the code or algorithm specific to our distributed application, we have made a choice of model in terms of the physical running environment and all surrounding operations.

Making this choice will inevitably shift the focus of our study from the system itself to a (simplified) model of it. Reducing the level of detail and increasing the level of abstraction in a model allows us to study different aspects of a system. However, since we disregard (perhaps vital) aspects of the system when we create our model we can only hope that properties valid in the model will translate to the real system. Such validation of results from the model is often desirable but can often be tricky. The modeled system can perhaps not be set up in such a way that make comparison meaningful or, if the system is only under design, be possible at all. In [19] the author suggests that the validation of model predictions should be made by domain experts such as other researchers, system designers, developers and others.

Different models of distributed systems have different level of abstraction depending on the properties of the system one might want to analyse. The levels

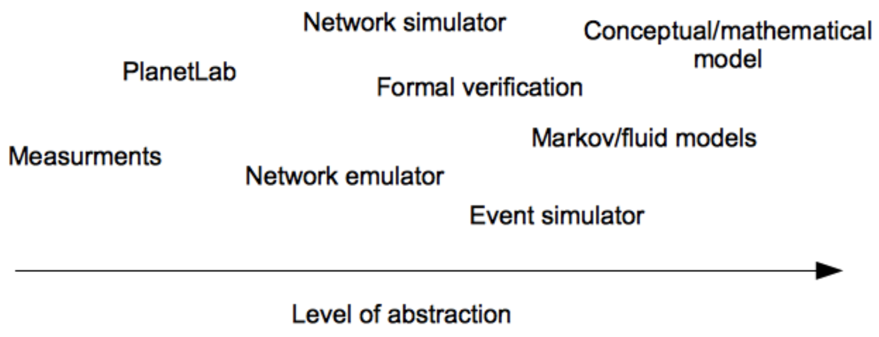


Figure 4.1: Different approaches to study a distributed system. The x-axis shows increasing level of modeling abstraction.

of abstraction used in different approaches to analyse distributed systems can be visualised in figure 4.1:

On the extreme left are measurements of running, live distributed systems and applications. In these studies very little is assumed about the running environment or the algorithms executing on the nodes. Measurements are made to quantify system behaviour with a minimum of interference in the process. Next are actual distributed applications running in some kind of controlled environment. It could either be on a well defined set of nodes such as on Planet Lab or a university LAN or a multi-core chip running a benchmark program in the lab. The difference from a live system is that the running environment is known and can partially be controlled. Also, the input to the system may either be generated by real users (people or other processes interacting with the system) or by the researcher conducting the measurement. Again, if input is not generated by live scenarios we need to state the model we use. The next level of abstraction is to emulate part of the system. An emulation is the process of replicating the execution of some process in a different environment. For instance, instead of sending a packet over an actual physical network link, the IP transport layer could be emulated with the assumption that the physical and data link layers are functioning as expected (without errors or with predefined error scenarios). In emulating parts of the physical execution environment we then inevitably need to make abstractions and model assumptions about the surrounding world. These assumptions are an input to the emulation process. In *simulating* a process we go even further in the level of abstraction. The difference between simulating and emulating a process is that the simulation is constructed upon a predefined (high level) model of the system whereas the emulation tries to mimic the process itself. Emulators allow capturing unforeseen network behavior

that cannot always be modeled in a simulator [12]. A simulation will only contain aspects of the system that we, by construction, put into it. We can for instance model a communication network by delaying packets a certain time before delivery. This is not how a real network behaves but can often be used as a good approximation.
On the next level of abstraction, is an event based simulation. In a discrete event based simulator (DES) all processes outside of the studied algorithm are modeled as sequences of events generated by our simulation environment. We then need to create models for all exogenous processes such as network communication, user interaction, arrival of tasks etc. Event based simulation therefore need various models for these processes. An alternative is to use series of recorded events (traces) and "drive" the simulation using these.
If we can model all the surrounding processes using some predefined description we can abstract away the whole physical running environment and only focus on the application layer algorithm. In doing so we have restricted the number of available states the system can be in. For some relatively small systems, in terms of number of nodes and complexity of the distributed algorithm, it is possible to formally verify the function and performance of the algorithm given *any* input [121]. Formal verification of distributed systems is a relatively new area of research that has previously been applied successfully to serial algorithms. A formal verification does not make any model assumptions about the algorithm itself but does not (generally) consider the execution environment explicitly.
For more complex algorithms it is not possible to evaluate all available states exhaustively. In this case we can make further abstractions about the algorithm itself and model only a subset of, or a collection of subsets, of all available system states. Such a state-reduced model however needs to specify how transitions are made between its states. For instance, if we want to model a complex large distributed application, a state of the model might be the number of nodes that has x number of entries in its routing table. The transition mechanism between two such states are in principle derivable from the algorithm itself (given the source code of the application) but will in many cases result in too many possible transitions for us to treat. There might be a large number of operations dealing with routing table entries but perhaps only a few of them have relevance for our particular analysis. We can then choose to focus on a subset of relevant operations and reduce the full algorithm to a *protocol* that can be studied in detail. Reducing the algorithm in such a way will inevitably shift the focus of our analysis from the algorithm itself to the protocol abstraction.

We have hence transferred our distributed system from an algorithm running in the physical world to a protocol operating on a state-reduced model. The protocol operates on the model by specifying how transitions are made between its states and thus we have created a *state transition* model of the actual system/algorithm. A state transition model can be formally verified for a limited number of nodes using formal methods [121], it could be emulated or simulated using some predefined input, or it can be analysed in a probabilistic manner. In a probabilistic approach we assume that the nodes reside in a particular state with a particular probability. The transition between states are governed by some stochastic rule or process which we can parametrize. This step further reduces the possible states of our model. Instead of studying actual executions of our (reduced) algorithm, we study probabilistic properties of it using parameters of some stochastic process as input. For instance, instead of providing our system model with recorded user patterns as input we replace the user behaviour with a stochastic process and study how the model behaves when we vary its parameters. We are then interested in determining the probability of observing the system in a certain state at a certain time.

One way to further simplify the complexity and state representation of the models is to exploit various symmetries of the system. It might be that certain highly correlated states can be bundled and modeled as one or that sub sets of the state space are independent which means that they can be modeled completely separately. One such simplification is to assume that the events driving the system are independent of each other as well as independent of time. In practice this is clearly not true. Requests for a particular piece of data as well as the number of joins and leaves in a system may depend on how many other requests there are, time of day, week and year. Assuming independence can however be justified as a fair approximation when the correlation among events is small or the driving processes operate on a time-scale much faster than the creation of correlations.

Looking again at 4.1 we see that the further to the right we move, the more we need to specify the model used in our analysis. How is the input to the system generated? Which states can the nodes be in? How do communication links work? That is, the further from the physical system we move, the more we need to specify the model in order for it to be scientifically sound.

4.2 Size and time scales

Another important dimension of abstraction is to study the system on different scales. A distributed system is on the one hand a collection of nodes acting according to a predefined protocol with certain properties (the microscopic scale). The system as a whole will then display (macroscopic) properties that is a function of these microscopic properties. How the microscopic scale generates a certain macroscopic behaviour is many times a non-trivial question and has been studied in other disciplines such as physics and biology for a long time. To use an analog from physics, if we want to study the temperature of a piece of matter we can either measure the kinetic energy of each of the participating atoms (analog to the left side of 4.1) or we can create a model of how temperature and volume (of mercury) are related and use a thermometer. The latter approach will provide an estimate of the average energy among the atoms (temperature) but will not claim to give a full state description of the system. The collective movement of the atoms is the systems microscopic scale that results in a macroscopic property which can be

measured, the temperature.
We can study a distributed system in an analogous manner. If we are interested in how the system behaves on average under certain conditions we might not need to study the full state of the system but rather a subset of the states relevant to our study. For instance, if we are interested in the break-up probability of a Chord ring, we may omit how the CPU on individual nodes performs addition operations. What is important in this kind of scaling abstractions is that the system in question consists of a large number of nodes as well as the existence of an *average behaviour*. According to the law of large numbers, a system consisting of a large number of nodes will most likely display deterministic average case behaviour.

Similar questions have been studied in other disciplines such as physics, chemistry and biology for many years. In physics the discipline dealing with probabilistic models of large systems is called *statistical mechanics* [114]. Statistical mechanics deals with finding thermodynamic (macroscopic) properties of large number of interacting (microscopic) particles. One of the tools used in analysing the dynamics of these systems is to formulate a so called *Master equation* for the system that contains the probabilistic description of the time evolution of the state of the system.

Time is an important parameter. A system might display a certain behaviour on a short time scale but a very different one if we study it for a long time. Systems during start-up behave differently from systems under stable running conditions or shut-down. Therefore it is important to state what time scale we are interested in before we start our analysis.
Even though no system will run forever, often we are interested in the long term stability of an application and hence assume the existence of a *steady state*. A steady state is a state which the system reaches regardless of how it started out, given enough time. If we can show that a system has a steady state we know that it will reach this state eventually and will stay there. The steady state can then be thought of as the true long term behaviour of the system.

4.3 Simulating state transition models

Once the model is constructed with a well defined protocol, underlying communication graph and driving stochastic processes the first step in evaluating its performance is to run it on a discrete event simulator (DES). The purpose of the simulation is to provide us with a experimental environment where we can test our assumptions about the system as well as monitor and study various types of dynamic phenomenon. The simulation environment of the system model has now replaced the physical environment of the actual system and is from this point on the entity we are trying to explain and evaluate the performance of. In many analytical works of system performance this crucial point is implicit. It is however important to keep in mind that the predictions made about the model need to be justified. If the simulated model displays some interesting behaviour it is not necessary for a deployed system with real clients to do the same.

The simulator and the simplified model can be thought of as a first step to trying to understand the behaviour of a complex system. When its predictions are tested against a real system the output should be used to enhance the model. This procedure should be continued until the system can be explained on a satisfactory level of detail that can be used as a starting point for the next iteration of system design and so forth.

The construction of the model and implementation of it in a simulator is a very important and non-trivial task. Many times we end up with a model that is simply too hard to treat analytically even approximately. In this case the simulation can give valuable insights in its own right. In this thesis, the modeling of the work stealing in the TileraPro chip for instance, does not have a natural steady state since the execution has a well defined start and end. The program executing on the nodes is also in many cases very complex and in order to draw some conclusions regarding design choices one needs to rely on a realistic simulation model. The same goes in part for our model of DHTs in the presence of network address translators. The resulting structure of the overlay networks becomes highly complex and small variations in sequences of events have a huge impact on the evolution of the system. In such cases we are not able to approximate the transitions as Markovian and are again reduced to studying the output of the simulator. One obvious caveat with the simulator is however that it has a limited capacity in terms of number of nodes as well as execution duration. To be able to study truly large systems we need to study the asymptotic behaviour when number of nodes grow very large. Our ambition is then always to try to find analytical expressions for system performance that hold for arbitrary system sizes.

4.4 Analysing state transition models

This thesis considers a set of state-reduced models for four different systems described in the previous chapter. In each of the analysis we consider a set of nodes running a common protocol on top of a connectivity graph. Each node is modeled by a set of mutually exclusive states together with a set of transition probabilities derived from the protocol. We model the input to the system from the external environment as a set of stochastic processes that effects the state transitions. Such a system description can more generally be described by a transition matrix $\mathbb{W}(t)$ whose matric elements describe the probabilities of transition between state i and j at time t. The mutual exclusivity of the states is important for this representation to be meaningful in a probabilistic sense.

So far we have not said anything about the nature of the transition matrix or the type of stochastic processes we use to model our external environment. Two important assumptions are made throughout our analysis, namely:

1. The transition matrix has finite dimension.
2. The stochastic process described by the transition matrix is a Markov chain.

The first assumption is important since it, if the markov chain is ergodic, will ensure that the system we study will eventually reach a unique steady state. What it means in practice is that we assume that there is a maximum number of possible states and number of participating nodes in the system. This is of course true in reality since there are only a finite number of computers with finite state representation capacity. In theory however this is not necessarily so. Some simple processes, such a random walk [95], will with some probability reach and exceed any positional state which means we need an infinite size transition matrix to define such a process. However assuming finite dimensions is a good approximation in our case since most of the processes we study have a limited number of states and the probability that we have an infinite number of states ($N \to \infty$) is exponentially small.
The second assumption means that our system (defined by our stochastic matrix) is not dependent on its whole previous history (all previous visited states) but only on its *one* previous state. That is, if we know the state the system is in right now, how it got there in the first place is irrelevant for the future evolution of the system. This may sound like a strong assumption but is in fact true for many systems in reality. If the system for instance has two ways of getting to position x and this has relevance for the future evolution of the system then we can just extend our state space to incorporate the two paths (state trajectories). The extended description will then contain information about how the system got to state x but will not be dependent on any previous state information not contained in the stochastic matrix. The art of modeling a system in this way is then to include enough state for the representation to be approximately Markovian and still keep the state space as small as possible. These assumptions limit the study of our state-reduced distributed system model to the study of finite dimensional, continuous time Markov chains (CTMC).

4.5 Methodology - how the analysis is performed

The study of steady state behaviour of CTMC's as a step in analysing computational systems is a well established technique [19].

Given that our above assumptions can be justified and we are able to model our distributed system as a finite state, continous time Markov chain we still need a recipe for how to transform the physical distributed system (left part in figure 4.1) intto a tractable model (right part). Also, given the model, we need to understand how to derive relevant performance metrics from it.
The first step in doing such a performance analysis is to determine which set of system characteristics we would like to study. It can be the breakup probability of

a Chord ring, the average run time of a multicore algorithm etc. These system characteristic(s) under study are dependent on the algorithm and dynamic behaviour of the underlying nodes (microenvironment).
Next we need to specify how the system under study is affected by its external environment. That is how does the system get its input? We could for instance be interested in how to best model the request rate for a piece of data or how to model the process of delivering a packet over a network link. All interactions amongst the nodes as well as all interacting surroundings need to be specified in the model. As mentioned above, when we simulate a system it is possible to use trace data to drive the system but in a purely theoretical analysis we typically need to use some probabilistic input.
Next we need to study the underlying algorithm and determine which parts of it are of relevance to the system characteristics under study. This subset of the algorithm is specified in a protocol that is the object of study. Such an abstraction step is very important since it reduces the set of potential states we need to consider. For instance if we assume that the underlying network layer can be approximated by perfect links and that network delays are negligible (done in the modeling step), or assume some process happens much faster than others, we do not need to consider parts of the algorithm related to setting up connections with handlers for various malfunctions etc. We can just replace the whole connection procedure of the studied algorithm by a simple "connect(node)" in the protocol.
Based on the questions of interest and the protocol we then need to define a set of system state variables from which we can deduce the system properties we want to study. For instance, if we want to study the break-up probability of the network, a suitable state variable might be the number of nodes that currently have an invalid pointer in their routing table. This modeling step connects the microscopic (protocol) description of the system with the macroscopic characteristics that we have set out to study.
Using the protocol we can next assign transition probabilities between the states. Two things are important in this modeling step. First we need to consider *all* possible events prescribed by the protocol and how they affect the values of our state variables. Secondly, the transition events should be *mutually exclusive* and sum of probabilities of all events, including the event that the state is unaltered, should add up to one. The next step is then to determine the probabilities for each possible transition during an infinitesimal time interval and thereby specify the evolution of the system.
The functional evolution of the full probability distribution of the state variables is in the physics context called a *Master equation* [114]. The master equation hence specifies the full probability distribution for the evolution of the system beginning from any initial condition. Many times, however, we are only interested in the long term stability of a distributed system and would like to know how it behaves when it is operating under stable conditions. In this case it suffices to solve the master equation under steady state conditions, namely we solve for the time independent probability distribution. Hence, provided there exists a unique steady state we can

solve for this by putting the time derivative of the probability to zero. This provides us with the long term stable value for our state parameters under the specified conditions.
The last step is then to calculate values for the macroscopic system parameters in steady state. We can simplify the analysis further by looking not at the full probability distribution but at the average value of the quantity of interest in steady state.

The methodology used here to analyse distributed systems can be summarised in these steps:

1. Select a set of endogenous *macroscopic variables* to be studied.
2. Define a set of external stochastic *processes* that drives the system.
3. From the algorithm under study, derive a *protocol* that is the part of the algorithm affecting the selected variables.
4. Define a set of system *states* from which the values of the variables of interest can be derived.
5. From the protocol, derive *transition probabilities* between the states.
6. Formulate a *steady state* relation and solve the relevant equations to get an average value (or higher moments).

4.6 Mathematical formulation

We model our distributed systems with a discrete event, continuous time Markov chain (CTMC) over a set of finite state machines $N(t)$ interacting over a network $G(E, V, t)$. Following the notation from [114] we denote the infinitesimal transition probabilities for the joint system state, $\mathbb{W}$. This matrix contains the probabilities that a transition occurs between any two states during an infinitesimal time Δt.

Ideally we would then like to state and solve the equations for the evolution of the full system state that is given by the Chapman-Kolmogov equation. In the continuous time limit, the Chapman-Kolmogov equation is equivalent to the *Master Equation*:

$$\dot{p}(t) = \mathbb{W}(t)p(t) \tag{4.1}$$

where

$$\mathbb{W}_{n,n'}(t) = W_{n,n'}(t) - \delta_{n,n'}(\sum_{n''} W_{n,n''}(t)) \tag{4.2}$$

Where n and n' are system states and $W_{n,n'}(t)$ are the transition probabilities from a state n' to a state n. In our case we study only *time homogenous* processes meaning $W_{n,n'}(t)$ is independent of time. The Master Equation describes how the probability distribution over system states evolves in continuous time and contains all information about the system dynamics. The general solution of the equation is

$$p(t) = p(0)e^{\int_0^t \mathbb{W}(t')dt'} \tag{4.3}$$

Getting a closed form expression for the time dependent solution of the Master Equation is in many cases very hard and often solved numerically or studied in the steady state limit of $t \to \infty$. In steady state the state occupation probabilites, p are time independent and given by $\mathbb{W}\pi = 0$ for some steady state probability vector π.

For a finite state chain such a steady state exists as long as the chain is aperiodic and irreducible. For an infinite size chain on the other hand we need to determine if the chain has a steady state by either finding a state vector for which $\mathbb{W}\pi = 0$ and $\sum \pi = 1$ or use the Foster-Lyapunov test [90] to see if the chain is positive recurrent. If so then a steady state exists.

In practice we have in this thesis implicitly assumed the existence of a steady state. This assumption can be justified by letting the chains have a very large but finite number of states for instance by restricting the number of users in a system. No real system can handle an infinite number of users so the approximation is relevant in most cases. Further many of the CTMC's studied in this thesis are linear functions of well studied birth/death processes that in turn are known to have steady state distributions.

4.7 Related tools

There are other approaches that model and analyse distributed computer systems as inherently stochastic. The methodology in these approaches overlap in part with the one used in this thesis. The similarities and differences of some popular related tools are summarised in this section.

Queuing theory

Queueing theory is a collection of tools for analysing the behaviour of queuing systems [67, 9]. It has had great success in analysing the performance of many networking systems from manufacturing lines to packet switch network protocols. Queuing theory deals with systems where a set of "customers" arrive to "service stations", get service and depart. Given the manner in which the customers arrive as well as the way service is given, queuing theory aims at analysing various properties such as average waiting time for the customers (system time), the probability of infinite queue length (stability) and more. Both time dependent, transient properties as well as long term steady state behaviour are studied in the literature.

The subject has been studied for more than a hundred years starting with the works of Erlang in 1909 [43]. In queuing theory there is a taxonomy for different types of queues denoted in the so called Kendalls notation [64] where the type of arrival process, service process and other properties of the queue are stated.

Given the general nature of a queue the techniques from queuing theory have also found application in the analysis of distributed systems. The peer-to-peer systems modeled in this thesis are for instance, in the language of queues a $M/M/\infty$ queue for the $N(t)$ process. There are attempts to extend the queuing theory taxonomy to include for instance peer-to-peer systems [82] but the focus in most of queuing theory is still to study systems which generate workload and service the same.

An explicit limitation in the scope of systems studied in general queuing theory is that they are fairly well approximated by one-step processes. These are the processes where the stochastic variable under study increases or decreases by a small finite number and the transition matrix hence has most of its weight around the diagonal. Some distributed systems depend however on dynamics where state transitions are made between non-adjacent states (there is a non-zero probability to move from state n to states other than $n+1$,$n-1$ or n) which then require other techniques. In this thesis we are interested in systems where the customers act according to an internal program and make active choices rather then just being moved around and getting serviced. Our models translate to rather complicated rules for how the system state evolves and are better modeled as general CTMCs than as queues.

The mathematical foundation of queuing theory is the study of stochastic processes and among them also continuous time Markov chains. The master equation is formulated in the queuing theory literature by Kleinrock [67] as a fundamental relation in the analysis of queueing systems:

$$\frac{dH(t)}{dt} = H(t)Q(t) \tag{4.4}$$

where

$$Q(t) = \lim_{\Delta t \to 0} \frac{P(t) - I}{\Delta t} \tag{4.5}$$

is denoted the *infinitesimal generator* of the process, P being the one-step transition probability matrix and I the identity matrix.

Comparing the two formulations, we see that they are equivalent and describe the time evolution of the occupation probabilities for a discrete state, continuous time Markov process. In the literature [9] treatment of CTMCs are generally considered to be a part of queueing theory. The analysis performed in this thesis is not classical queueing theory however due to the very different nature of systems and questions studied which impels us to model our systems using first principles, rather then mapping our systems to well known models of queues.

Fluid models

One technique from queuing theory that has found application in modeling large complex distributed systems is *fluid modeling* [68]. A fluid model is a deterministic version of the stochastic system in that explicitly stochastic processes of variables such as the arrival of customers, service time etc are replaced by their deterministic counter parts as a function of time. Mathematically we replace the underlying stochastic process by an ODE and abandon our probabilistic description of the system. If done correctly the hope is that the solution to the deterministic equation will match the long term behaviour of the stochastic process.

The fluid model can be interpreted as the mean flow of the stochastic model. [89]. Whether or not the fluid model strictly corresponds to the mean behaviour of the stochastic process or whether the fixed point of the ODE matches the stationary behaviour of the stochastic process needs to be determined case by case and is often non-trivial to derive [20]. Intuitively these kinds of approximations can be justified however, if the variation of the process is small compared to its average value. This is true if there exists a time scale over which there are so many events that the law of large numbers (LLN) can be used to characterize the process by its average value during the studied interval. This average value changes on a macroscopic timescale, t which is then used to describe the dynamic evolution of the system. The notion "fluid" refers to this discrete to continuous transition. Instead of considering individual events, changes in the flux of events are studied. In the theory of queues, the fluid limit is motivated in the case of "heavy traffic" [68] which means that the length of a queue, and therefore the waiting times (time spent in the queue) are much larger than the fluctuations in the arrivals of new customers. Such a system can then be fairly well approximated by its fluid limit.

For instance, if we want to model a distributed system in which each node performs some task modeled as a stochastic process with rate σ, much higher than the rate λ at which nodes join and leave the system then we can model the system in the *fluid limit* whereby the number of tasks performed are characterised by their mean rate $\rho(t)$. The total number of events in the system per unit time can then be approximated by $N(\lambda, t)\rho(t)$, where N is the number of nodes in the system at time t and is given by a stochastic process with rate λ. The stochastic process with parameter σ is now replaced by a deterministic process fully described by $\rho(t)$.

One caveat with using a fluid model is that since only average values are studied, effects caused by variation in the process such as a worst case analysis or first passage problems cannot be analysed. If we want to know the probability of congestion in a network link for instance, a fluid model can only provide a lower estimate. This is because congestion occurs at the first instant that a communication channel is full, which will certainly happen well before it is full on average. The intention of a fluid model is however in many cases to generate an initial understanding of the system rather than to be used as a tool to understand it in detail. "A fluid model is often a starting point to understand the impact of topology, processing rates, and external arrivals on network behavior." [89]

Fluid models differ from rate equations in that the rate equation describes the evolution of the first moment of the master equation whereas the fluid model describes the time evolution of a deterministic variable. Even though the rate equation describes the time evolution of the mean (and so does the fluid model), the rate equation is derived from the infinitesimal transition rates of the full probability distribution (master equation) of the underlying stochastic process, which contains all statistical moments, whereas the fluid model simply replaces the underlying stochastic variable with a deterministic variable as a function of time.

Mean field models

In many systems the assumption that the states of the nodes are independent is too strong. In queuing systems for instance, we might have a network of queues where the output from one queue is the input to another. In this case we cannot (even approximately) assume that the state of the two queues are independent. A frequently used method to deal with such *coupled* systems in the physics literature is to study systems in the so called mean field limits. If we think of the nodes in our system as a set of interacting particles then the analog to physics becomes most clear. Instead of considering how the state of a particle changes with all possible kinds of interactions with other nodes, which in theory might depend on the full state of the whole system, we approximate the environment surrounding a particle by a *mean field*. The notion comes from physical systems where a particle n interacts through a field characterised by a potential function $V_n(\bar{x})$ where $\bar{x}$ is the full state of the system. Instead of solving the state equations for the whole $\bar{x}$ we consider interactions of the type $V_n(<x>)$ where $<x>$ is the "mean field" of all other particles.

Constructing a mean field limit in a networking system can be done in a similar way [18]. The first step is to decouple the system into N fully symmetrical units. Next we select (or "tag") one unit and study its interaction (as state transitions) with the other $N-1$ units. Finally we let $N \to \infty$ and hope that the transition probabilities converge to a well defined limit, the mean field limit.

Even though this may sound straight forward, the construction of a model and assignment of suitable units and interactions which are well defined in the large N limit is non-trivial.

In this thesis we have not constructed explicit mean field limits but have instead studied the systems in steady state which is the $t \to \infty$ limit, keeping N fixed. Some of the resulting equations have an explicit N dependence which lets us study the large N limit as well but this is not a mean field limit strictly speaking.

General mathematical models

There are many attempts to analyse distributed systems using various handcrafted probabilistic models, which correspond to the very right of figure 4.1. Often there is no explicit methodology applied in the modeling process since one is rather interested in evaluating high-level conceptual properties of a potential system design. Although this approach does not qualify as a separate method some set of tools, like Chernoff bounds, moment methods and Martingales, are commonly more used in the analysis of general probabilistic models (see [92] for a general introduction). A few related examples include; Establishing message and time complexity for a generic one dimensional greedy lookup using analytical bounds on Markov processes [10] or bounding time complexity for the averaging time of a gossip algorithm [21].

Formal verification

Formal methods refer to a set of mathematical modeling tools used in the design, development and verification of both software and hardware systems [121]. The idea is to specify a system or application using a formal syntax and let a computer program prove properties of the state reached by the program in execution. Ideally one would like guarantees that the system behaves as expected regardless of input and execution paths. One would then be able to spot subtle bugs which can only occur under rare circumstances. In the formal method language such an approach is known as model-checking. Given a mathematical model we would like to exhaustively explore all possible states and make sure all of those are valid in terms of our high level specification. Formal methods and model-checking have been successfully applied to verify even large programs such as operating systems [66]. In recent years formal methods have been applied even to distributed systems and peer- to-peer applications [124, 11]. One caveat however is that since systems with dynamic membership (such as peer-to-peer) can in principle contain any number of nodes and thereby any number of states, a formal description would have to be very large (known as the state explosion problem [52]). One remedy for this is to state the formal model on a higher level of abstraction using domain knowledge and generate a so called lightweight formal model. Such a lightweight model is similar to the models studied in this thesis. Formal methods have been successfully applied in this way to distributed systems. In [124] a lightweight formal model of Chord is used to prove that the system is in fact not self-stabilising. The self-stabilising condition means that the system, from any initial state, will, left by itself end up in a state which is "correct" according to a global specification. This state should be reached within finite time. The initial claim that Chord recovers from any initial bad configuration has hence proved to be false.

Such conclusions are very hard to reach without exhaustively generating all combinations of possible events and verifying that they are legal. Other examples of formal verification and DHTs include [11] where the correctness of Chord is studied using a process algebra model. An inherent problem however is that even for simple models, there is no way one can study systems of Internet wide sizes with millions or even billions of nodes.

4.8 Methodological limitations

Even though the method used in this thesis is very general it has some caveats and limitations when used in practice. First of all, creating the state transition model from the physical system is sometimes the hardest part of the analysis and does not really follow a general procedure. The set of macro variables that are selected and how they are related to the microscopic states is not universal, meaning the method is not guaranteed to output the same model when applied to the same system. How many simplifications and reductions that are made is partly a subjective choice. This is an inherent problem in all modeling processes and modeling is sometimes more like an art than a science.

Given that our choice of macro variables can be justified, the assumption that our system can be modeled as a Markov chain can sometimes be questioned. Many external processes depend heavily on their state trajectories and the number of states needed in order for it to be even approximately Markovian is too large to handle. For instance the state of execution of a large, complex program will depend on each operation previously performed and does not easily reduce to a set of macroscopic state variables. There is a risk of reducing the system too much which will result in the output of the analysis being misleading.

Provided that we can capture the set of universally important macro variables and state our model it is often not straight forward to get a closed form expression for state evolution. Many times the Master equation is solved numerically which will then introduce additional choices of how and if the equations can be simplified, which solver to use and how it is parametrized etc.

Another inherent problem when we have to make abstractions in the modeling process, is how to verify our results? Many times it is not possible to design an experiment for a real system which will verify or falsify the predictions of our model. We are then left to compare the outcome of our analysis with abstractions of the real system like emulations or simulations running either the application algorithm or the reduced protocol.

Another issue of studying average case behaviour is that many times rare critical events might have a large impact on system behaviour. There is no straight forward way to extend the method to study worst-case behaviour for instance. In many cases we would like to know the probability that a certain rare event will or will not take place. Even though our model description is very general and our method does not restrict analysis to steady states we need to complement our method with other tools to study critical events such as Chernoff bounds and/or large deviation theory.

Further, no physical system resides in true steady state. Eventually all systems will decay and the system state will change in one way or the other. However, our assumption of distributed system being in steady state does not mean that it will last forever. What we assume is the existence of a state which will last long enough that all variations in the macroscopic variables we are studying are small in comparison. Even such an assumption of separation of time scales can in many cases be questionable. Can we assume the popularity of content is more or less constant over the time that it takes for a large number of people to request it? Can we assume that the join and leave rates of users to an application remains constant over several hours/days/months? In many cases we use the steady state assumption as a benchmark starting point for further investigation of a model. If we know how the system behaves in steady state we have something to compare other scenarios with.

Chapter 5

Thesis Contributions

This chapter contains a summary of the contributions made in this thesis. It starts with a list of publications by the thesis author followed by a detailed of the contributions made in the work of each system.

5.1 List of publications

List of publications included in this thesis

1. J. Ardelius, et al. *On the effects of caching in access aggregation networks.* Proceedings of the second edition of the ICN workshop on Information-centric net- working. ACM, 2012.

2. J. Ardelius, and B. Mejias. *Modeling the performance of ring based DHTs in the presence of network address translators.* Distributed Applications and Interoperable Systems. Springer Berlin Heidelberg, 2011.

3. J. Ardelius and S. Krishnamurthy. *An analytical framework for the per- formance evaluation of proximity-aware overlay networks.* Submitted. Technical report can be found at, Tech. Report TR-2008-01, Swedish Institute of Computer Science.

4. KF. Faxen, and J. Ardelius. *Manycore work stealing.* Proceedings of the 8th ACM International Conference on Computing Frontiers. ACM, 2011.

5. S. Krishnamurthy, et al. *Brief announcement: the accuracy of tree-based counting in dynamic networks.* Proceedings of the 29th ACM SIGACT-SIGOPS symposium on Principles of distributed computing PODC. ACM, 2010.

List of publications by the thesis author not included in this thesis (in reverse chronological order)

6. J. Ardelius and E. Aurell. *Behavior of heuristics on large and hard satisfiability problems.* Physical Review E, 74(3):037702, 2006.
7. J. Ardelius, E. Aurell, and S. Krishnamurthy. *Clustering of solutions in hard satisfiability problems.* Journal of Statistical Mechanics: Theory and Experiment, 2007:P10012, 2007.
8. M. Alava, J. Ardelius, E. Aurell, P. Kaski, S. Krishnamurthy, P. Orponen, and S. Seitz. *Circumspect descent prevails in solving random constraint satisfaction problems.* Proceedings of the National Academy of Sciences, 105(40):15253, 2008.
9. J. Ardelius and L. Zdeborova. *Exhaustive enumeration unveils clustering and freezing in the random 3-satisfiability problem.* Physical Review E, 78(4):040101, 2008.
10. J. Ardelius. *On state space structure and average case complexity in random k-sat problems.* 2008. Licentiate Thesis, TRITA-CSC-A 2008:08.

5.2 Contributions

DHT under link delays

In this work we set out to analyse the performance impact of finite link delays on the DHT Chord. The work is an extension of previous work on Chord under node churn [109]. The question we tried to answer was how the routing performance was affected when network messages could not be expected to be delivered instantaneously. We found that Chord is not able to perform lookups in O(log(N)) time using the finite link delay model. We further suggest how the model can be used to analyse various proximity-aware routing policies.

My contribution to the work was to implement the model in a discrete event simulator (the previous work used a Monte Carlo simulator which did not use the notion of time). I did some of the calculations in the appendices and generated all the numerical data with the simulator.

DHTs and NATs

In this work we studied the performance impact of network address translators (NATs) to the DHT Chord. An implicit assumption up until this time in most analytical work, was that the underlying network is transitive. In the presence of NATs however, these assumptions do not hold. We therefore constructed a model of a simple NAT network where nodes could either have unique open IP addresses or sit behind a NAT. We studied how the fraction of nodes behind NATs affect the overall performance of Chord in terms of routing hops and correctness.

Ardelius, John, and Boris Mejias. "Modeling the performance of ring based DHTs in the presence of network address translators." Distributed Applications and Interoperable Systems. Springer Berlin Heidelberg, 2011.

My contribution to the work was the construction of the model together with Boris Mejias. I did the entire implementation of the model and the analysis.

GAP

Here we studied the performance of a distributed Bellman-Ford protocol known as GAP. It is a fully distributed algorithm used to estimate the number of participating nodes in a system. It can also be used to calculate averages and sums of variables distributed among the nodes. In this work we analysed the impact of node churn on the reliability of the estimates of the algorithm at a root node in the network. To do this, we constructed a continuous time Markov chain model for the number of nodes at various distances from the root node as well as the value of the local estimates as it propagated through the network.

Krishnamurthy, Supriya, et al. "Brief announcement: the accuracy of tree-based counting in dynamic networks." Proceedings of the 29th ACM SIGACT-SIGOPS symposium on Principles of distributed computing. ACM, 2010.

My contribution was to construct part of the model for the number of nodes at particular distances from the root together with Supriya Krishnamurthy. I implemented the model in a discrete event simulator to verify the analytical outcome of the model. I made some of the numerical calculations in the paper.

Caching

In this work our aim was to understand the effects of implementing an in-network caching hierarchy in an access aggregation network. The work tries to analyse the impact of the idea of information centric networking (ICN). If data is the basic element of a network, caching becomes key but how can we build an efficient caching system? To answer these questions we constructed a network model of a hierarchical access aggregation network with continuous requests for various pieces of content. Using the model we were able to analyse the impact of various caching policies such as least frequently used (LFU) and least recently used (LRU). We were able to provide accurate approximations to the performance of these policies when the number of available items is large but finite and caching capacity is restricted. Using a Pareto distribution for the request rates we further studied the impact of these policies in various network topologies.

Ardelius, John, et al. "On the effects of caching in access aggregation networks." Proceedings of the second edition of the ICN workshop on Information-centric networking. ACM, 2012.

My contribution was creating the model. I did all the implementation work as well as all of the analysis. Together with Bjorn Gronvall, I set the parameters of the model to values that are reasonable in practice.

Manycore work stealing

This work studied the scaling properties of a generic many-core processor architecture under the performance of the work stealer Wool. The main focus of the paper was to device the applicability of a nested task parallel to the new TILEPro64 chip. Wool uses a work stealing heuristic to distribute processing load among all the available cores. Apart from a detailed benchmark study, we also constructed a model of the work stealing dynamics using a set of uncorrelated stochastic processes. Using the model we were able to simulate the behaviour of a work stealer running on thousands of cores.

Faxen, Karl-Filip, and John Ardelius. "Manycore work stealing." Proceedings of the 8th ACM International Conference on Computing Frontiers. ACM, 2011.

My contribution to the work was, together with Karl-Filip Faxen the construction of the stochastic model. I also did the implementation of the model in the simulator and summarized the results from it.

Chapter 6

Conclusions and Future Work

This chapter overviews the contributions of the work done in the thesis as well as the lessons learned in the process together with a general discussion on the outreach of our methodology, including future directions.

6.1 Modeling the lookup latency of distributed hash tables in dynamic environments.

Our study of distributed hash tables and Chord in particular has revealed several interesting system properties. From a purely theoretical perspective, we have been able to show that the average case routing guarantees of Chord and related systems, to be able to deliver a message in $log(N)$ number of network hops, does not hold in the presence of link delays. Even a very simplistic delay model for the network links will cause latencies which will grow faster than $log(N)$ when the system size increases. Apart from this result we have also been able to predict the impact of various proximity aware routing policies when applied to Chord thus providing an analytical framework for comparing different design choices.
Further, we have also studied the impact of introducing non-transitive links in the Chord network model. This was motivated by the widespread usage of network-address- translators in the underlying network topology. We show that non-transitivity causes both large lookup inconsistencies (several nodes owns the same key) as well as large heterogeneity of load among the participating nodes. This can be remedied by restricting access to the system for nodes with non-transitive connectivity but will then reduce the benefit of the system as such.

6.2 Content distribution using hierarchical caching in access aggregation networks.

We develop an analytical model of a large scale access aggregation network receiving a continuous stream of requests from external clients. We calculate analytical expressions for the performance of LRU and LFU caches and use these to derive probabilities that a given item resides in cache, determine cache efficiency measured in terms of hit rate as function of network load, data availability and more. The model enables us to study systems scaling to millions of available data items and thousands of nodes, something which is very hard or impossible using simulations. Our results show that the more computational and memory expensive LFU eviction policy gives on the order 10% better hit rate performance, for reasonable cache sizes, as compared to regular LRU and this number decreases logarithmically to zero as the number of available data items grows large. Although our model is not restricted to any particular data popularity distribution, we have focused on Zipf distributed popularities in this work. One observation using this distribution is that the LRU cache performs very poorly for very large amounts of data items compared to LFU that provides hit rates of a few percent higher even for a very large amount of items and finite cache size. The result implies that for practical purposes, frequency estimation is a very important and is perhaps the key component of an efficient eviction policy.

6.3 Performance evaluation of multi-core work stealing heuristics

Motivated by the current fast paced increase in number of CPU units per processing chip we created a model of the computational environment of the TILEPro64. The model was used to capture the effects of deploying different work stealing protocols in the run-time architecture. By using the model we are able to predict the performance of a system with an increasing number of cores and make predictions for systems much larger than those existing today. More specifically we looked at which property is more crucial for overall performance when nodes steal tasks from each other and how many tasks there are in the task pool relative to how deep we are in the computation tree (which measures how many times the task has been subdivided into two parallel tasks). These two measures are used by a node when sampling which other node to steal from. These approaches were compared to the strategy of not sampling at all but rather picking the first task available.
The results lead us to conclude that both strategies are similar in performance but much more efficient than not sampling. However, they come with some computational overhead which needs to be considered in an implementation.

6.4 Reliability estimation of a distributed Belman-Ford protocol.

Using our model of an ad-hoc system running the Belman-Ford protocol we are able to predict the performance in terms of estimation accuracy at the root, with very high accuracy. Given a Poisson churn model the model predicts the ratio of the number of nodes in the system estimated by the root in relation to the true value. This metric can be calculated for a wide range of system sizes, churn levels and node-to-node connection probabilities. Further, the analysis poses some interesting conjectures for the dynamical properties of the system. The number of nodes leaving a certain level (given by the distance from the root), moving either closer to or further from it seem to be exactly balanced by the number of nodes from other levels entering this level. If this conjecture turns out to hold (which our simulation results support) it would be an important result in the theory of network queues.

6.5 Lessons learned

If one wants to predict how systems, containing millions of participating nodes, behave in dynamic environments there are few tools available today. Formal verification is perhaps the most reliable but computationally most expensive tool which does not let us study very large systems. Simulations and emulations let us study more realistic applications but do rely on good models for user behaviour and other external interactions. Again there is a limit to the number of nodes we can study by these methods, and we may in any case only be able to sample subsets of the full parameter space. Highly conceptual models might be mathematically tractable and give us valuable insights but might be too simplistic when it comes to modeling the running environment. Hence whether results for the model translate to the real world is not always clear. By studying rather extensive protocols, analogous to application algorithms, together with explicit stochastic processes driving our models we are able to study a wide range of large scale systems and applications. By giving up a little bit of mathematical rigour and simplifying algorithms and processes a little we are able to draw interesting conclusions not only about particular systems but of classes for many systems for truly large systems sizes.

As mentioned in the previous chapters, the various tools discussed have different working ranges, different pros and cons and should be viewed as complements rather than alternatives. Our impression though is that the distributed systems community has lacked the methodological approaches used in other natural sciences that also study large scale systems. By applying these tools to the set of systems in this thesis we have showed that there is a place for particle models in performance analysis of distributed computer systems.

6.6 Future directions

A very promising direction of research is to combine the approach taken in this thesis with application of lightweight formal methods [124]. The lightweight model can then be checked formally for a restricted number of nodes. This would justify the assumptions made in the asymptotic analysis for large system sizes.

As for the extensions of the analytical modeling, independence among events in the driving processes is many times a quite strong assumption. In order to study how correlation will effect these systems one would need to alter the models and either let the parameters of the driving processes depend on endogenous variables or to incorporate explicit correlations in the transition probabilities. Either approach will complicate analysis but will give a more realistic model.
Another interesting line of questions relates to transient behaviour in distributed systems. User patterns such as flash-crowds, and other types of traffic bursts are not included in the steady state description of the system but are very relevant to their performance. Some systems like the single execution of a multi-core program does not in practice reach a steady state but displays different properties during its run-time. Such behaviour is possible to study using the same methods described here but with the difference that the we do not sample or study the system in steady state but keep the time dependence explicit. Another interesting direction is to modify our approach to study first passage problems or worst-case results using for instance large deviation theory.

Chapter 7

An Analytical Framework for the Performance Evaluation of Proximity-aware Structured Overlays

John Ardelius[1] and Supriya Krishnamurthy[1,2,3]

[1] Swedish Institute of Computer Science (SICS), Sweden
`[supriya,john]@sics.se`

[2] Department of Physics, Stockholm University, Sweden

[3] School of Computer Science and Communication, KTH, Sweden

Abstract

Predicting the performance of large scale distributed systems is a an important and non trivial matter. Understanding the interplay between network condition, user behaviour and application is key in designing and dimensioning networked systems. In this paper we analyse the performance impact of network delays in combination with user churn to the DHT Chord. We create a stochastic model of the dynamic environment and make predictions about lookup latency and routing table correctness. Comparison with simulation data show that our model is able to predict these properties with very high accuracy. We further show how the model can be extended to incorporate various routing policies such as PNS/PRS and how Chord scales as the number of users grow very large.

7.1 Introduction

DHT's have, in recent years, moved from being an academic research topic to being implemented in many large scale live systems [1, 2]. The fact that they are able, at least in principle, to route messages in $O(\log(N))$ number of network hops despite churn, where N is the average number of participating nodes, makes them a very promising tool for building large scale systems. However performance metrics from actual systems show that response time for key lookups is in many cases unacceptably large. In [30], the Kademlia-based DHT used for millions of nodes in BitTorrent is examined using trace data analysis and it is shown that some lookups take over a minute to complete. This is due to the fact that a large fraction of the nodes have dead connections in their routing list (inefficient stabilisation policies) as well as poor network conditions (large ping round trip times). Hence not only do DHT's need to survive churn, they also need to cope with varying network delays.

There have been several key papers that address the issue of churn in a DHT [81, 103, 50] and which suggest design-related tradeoffs that mitigate the effects of churn. Earlier work [53, 103, 31] has also anticipated problems due to network delays and various selection policies have been suggested to remedy this. Two such commonly used policies are proximity route selection (PRS) and proximity neighbour selection (PNS) whereby nodes try to either route queries through "fast" links or only select network neighbors which are "close". The issue of the interplay of churn with network-delays is also addressed [103] suggesting that low-churn and high-churn situations might require different latency-reducing tradeoffs. However a single analytical framework within which one might address these different design decisions as a result of the interplay between user-churn and network-induced delays is lacking to the best of our knowledge.

In earlier work [73], we have studied the Chord DHT [109] under churn, and have demonstrated the detail with which we can predict the average lookup length at any value of churn and system size. In this paper we extend the analysis to also dealing with delays in the network layer. The theoretical analysis of this extended model, when compared to simulations, gives a surprisingly accurate picture of the interplay between network churn and delays, for any system size, and all values of churn and system delay that are practically realizable. The analysis is easily extendable to addressing various locality-aware schemes such as PNS and PRS, or comparing design choices such as iterative vs. recursive lookups. A very interesting insight that emerges is that the combined effect of network delays, a periodic recovery scheme, and churn, results in lookup latencies scaling as $O(N^{\gamma}\log(N))$ as soon as there is any average node-to-node delay at all, instead of $O(\log(N))$. For low churn or small network delays that are negligible when compared to average node lifetimes, this change is hardly noticeable due to the low value of γ. On the other hand, for larger values of churn, larger values of delays or when the number of participating nodes grows very large, this change of scaling can lead to a large fraction of dead finger-table entries for every node in the network, and hence very inefficient lookups.

7.2 System model

We are interested in analysing the behaviour of the Chord protocol, as specified in [73], in the presence of node churn as well as finite link delays. To enable us to do this we extend the stochastic model from [73] to include the effect of delays in the underlying network layer.

Chord

Due to space limitations, we provide only a very brief description of Chord in the following section, with a view to explaining the parameters involved. A complete description of the Chord protocol that we use can be found in [73].
The Chord ring is a logical circular key space of size $\mathcal{K}$ populated by N nodes. To maintain the ring each node keeps a list of the s nodes succeeding it (according to local knowledge) as well as a pointer to the prececeeding node.
Routing using only successor pointers is not efficient and to reduce the average lookup path length, nodes keep $\mathcal{M} = \log_2 \mathcal{K}$ pointers known as the "fingers". Using these fingers, a node can retrieve any key in $O(\log N)$ hops [109]. The fingers of a node n point to a key an exponentially increasing distance 2^{i-1} (where $i \in 0 \cdots \mathcal{K} - 1$) away from n. The corresponding entry in the finger table is the first successor of this key.
As in our earlier work, we assume that arrivals and departures from the system are uncorrelated among nodes, time independent and given by independent Poisson processes. Node churn can thus be parametrized by a rate λ_j of joins for the whole system and a rate λ_f of failures per node.

To keep the pointers up-to-date in the presence of churn, we consider a periodic stabilization strategy and define λ_s as the (time-independent) rate at which stabilizations are scheduled, per node. With probability α, a node chooses to stabilise a successor and with probability $1 - \alpha$, it sends out a lookup message for any one of its fingers. We set $\alpha = 0.5$ in all that follows without loss of generality.

As an optimisation strategy that we implement, lookups are only sent out for fingers which are not the first successor. The first successor as well as all fingers which have the first successor's id in their finger table are corrected by the simulation environment without having to make a stabilization call. This enables us to study the effect of extreme churn rates or very large delays.

We are interested in calculating all quantities of interest in the steady state, where $N\lambda_j = \lambda_f$. In the analysis it is useful to define the ratio $r \equiv \frac{\lambda_s}{\lambda_f}$. One can think of $1/r$ as the time between two succesive stabilizations in units of node lifetimes. Alternatively r is the average number of stabilizations a node is able to do in its lifetime. For e.g., the value $r = 50$ means that a node is able to send out in average 50 stabilization calls in its lifetime. If we know that the node has an average lifetime of half an hour, $r = 50$ implies that it sends out a stabilization call every 36 seconds. In our analysis (as in earlier work [73]), we only need the ratio r as a measure of churn in steady state. Clearly if r is large churn is low, and vice-versa. We consider values of r ranging from 50 upto 1000 in our simulations. We have even looked at more extreme cases such as $r = 10$, where the optimisation procedure mentioned above is necessary in order to keep the ring connected.

Network model

We are interested in analysing a model where messages sent across overlay links are not necessarily delivered instantaneously. Here we assume that the transmission delay is mainly caused by delays in the network rather than by slow processing of the query locally.

To incorporate network delays in the stochastic model *e*ach message sent over the underlying network layer is delayed by an amount of time that is distributed according to an exponential distribution with a time-independent parameter λ_d, independent of the delays of other messages. This is a widely used base line model for packet based traffic in the networking community [68]. Further, no message is lost (except in the rare case that all the references in a nodes finger table and successor list are dead). Taken together, our link model implements the perfect delay link abstraction from [51] which is widely used in the analysis of distributed systems.

The inverse of the transmission rate gives us a typical time between a message being sent by the sender and received by the recipient. A typical latency of an internet link is around 100ms[54]. However, measurements of delays in DHT's [30] show that round trip times can range from 0.5 seconds upto several seconds or more (though this might be due to queueing and congestion which we ignore in this analysis). We could hence consider values of $1/\lambda_d$ to lie in this range.

However, since our unit of time is node lifetime, the relevant parameter that quantifies the delay in the network is again a ratio : namely the ratio of the transmission rate to the failure rate $d \equiv \lambda_d/\lambda_f$. This means that a node-to-node delay of a 100ms for a system of nodes that have an average lifetime of 1 minute is equivalent to a node-to-node delay of a second for a system of nodes that have an average lifetime of 10 minutes. Since these two systems have the same value of d $(= 600)$, their performance will be exactly the same, all other parameters being equal. Large d implies low delays and vice versa. In our simulations we have considered values of d ranging from 50 to 200. The case $d = \infty$ corresponds to the system without any delays studied earlier [73]. We only consider $r \geq d$ in our simulations as otherwise the number of unanswered queries keep increasing over time and do not reach a steady state.
The important parameters which we vary in simulations are hence N, d and r.
Under these assumptions we can model the quantities of interest in the Chord system- the fraction of dead fingers as well as the average lookup lengths for each finger-type - as Markov processes operating in continuous time. We specify the transition rates for these processes in terms of the parameters defining our system and calculate the steady-state values of these quantities. To validate our results we have also implemented the Chord protocol in a discrete event simulator with join,

leave, stabilisation and transmission events generated by Poisson processes with different rate parameters. We let the simulations reach steady state before comparing the recorded quantities with our theoretical predictions. For completeness we present a reference list of model variables:

λ_f	Rate of fail events per node.
λ_j	Rate of join events per system.
λ_s	Rate of stabilization events per node.
λ_d	Rate of message transmission per transmission event
r	Churn ratio, $\frac{\lambda_s}{\lambda_f}$
d	Delay ratio, $\frac{\lambda_d}{\lambda_f}$
N	Average number of nodes, $\frac{\lambda_j}{\lambda_f}$
$\mathcal{K}$	Number of keys in the system
M	Number of fingers per node, $= \log_2(\mathcal{K})$
α	Fraction of finger stabilisation rounds.

7.3 Related work

Proving performance bounds in the face of continuous churn is a challenging task. In [83], the authors prove a lower bound on how much bandwidth the maintainance protocol needs to consume in order for an N-node dynamic P2P network to remain connected with high probability. In our language this translates roughly to a lower bound on the value of r of the order of $\sim log(N)$.

Of much greater relevance to this paper are the results in [10], where bounds are derived on the delivery time of greedy routing strategies in the ideal case as well as when a fixed fraction of nodes or links fail. On the face of it, this result is not valid for continuous churn. However, as it turns out from our analysis, in a system without delays, continuous churn does lead to some fixed fraction of links failing in the steady state, with this fraction depending continuously on the parameter r. With delays, we find that the probability that a link is dead depends on how far away the link points to. Nevertheless, even in this case, the result in [10] is very relevant to us and we devote section 7.7 to understanding the implications of their bound to the analysis done in this paper.

Performance analysis involving predicting average long term behaviour under churn, is, done for example in [123] and in our previous work [73]. In [123], the focus is on building a model of churn that captures the heterogenous behaviour of end-users. This is done by viewing each user as an alternating renewal process with two different distributions for online and offline durations. Despite the fact that there are as many such distributions as users, the authors show that the lifetime of joining users, is given by one single distribution obtainable from the above. A result of relevance to us is that both the superposed arrival process as well as departure process converge to a homogenous poisson process when the number of users is large, hence motivating the simpler view of churn we use earlier [73] as well as in this paper. The model in [123] can also be generalized to include search delays. However, unlike in this paper, the authors are primarily interested in understanding the effect on node isolation. In addition, search times are given by a generic distribution and not a result of an explicit lookup process as in this paper.

To our knowledge, we do not know of any work which analytically derives performance bounds, or predicts average performance under both continuous churn and persistant network delays, at the level of detail which we develop in this paper. The model extended in this work was originally presented in [73] where churn in the Chord system was modelled using a poisson-arrival process with an exponential lifetime for nodes in the network. To maintain the ring in the face of churn, each node also performed stabilizations as explained in the previous section. By the use of the theory of continuous-time Markov processes, also called Master Equations [113] in physics, macroscopic metrics such as the average lookup cost and routing table correctness were calculated given all details of the Chord protocol. In the work presented in this paper, we generalize the above model, again keeping all details of the protocols, to understand how performance is affected by network delays. We also demonstrate the flexibility of our analysis by considering cases when the basic Chord lookup or finger-selection strategy is enhanced by proximity-aware routing or neighbour-selection schemes or lookups are made recursive instead of iterative.

Our modelling approach is related to the so-called "fluid models" [68, 27] in that the rate equations we write for the expected value of quantities of interest, are similar to equations one might get from the more deterministic approach common to fluid models [27], where events that happen on a much faster time-scale than the time-scale of interest are replaced by an average value. However we do not ever need to assume that any of the processes which lead to stochasticity in our model happen on a much faster or slower time-scale. Infact the most interesting effects arise when the different processes happen on similar time-scales. Hence our approach arises from a truly microscopic model, and though we only look at average quantities, in principle, we could also use the same approach to study fluctuations.

7.4 Analysis

In our analytical model, we have a set of interleaved independent stochastic processes - nodes join, fail and forward queries for themselves or for others. In addition, each of these queries is independently delayed by the network. Our final aim is to take all these processes into account in determining how the steady state value of the average lookup latency is affected when delays are present. We would also like to predict how performance (as indicated by the steady state lookup latency) scales with system parameters such as system size, stabilization rate and delay. In order to do this, we first need to compute the steady state fraction of dead kth fingers (*i.e.* the fraction of nodes whose kth finger points to a departed node). We denote this fraction $E[f_k]$. This is an important quantity since encountering dead fingers en route increases the lookup latency. The fact that lookup time is heavily dependent on the probability of dead nodes in the routing table is shown in measurements of real DHT based applications [30]. In our previous study [73] we have calculated $E[f_k]$ for Chord with churn but without any network delays. To calculate $E[f_k]$ with delays, we demonstrate in what follows that we actually also need to know the average latency of the lookup (denoted $E[c_k]$) for the kth finger. Since this latency depends on $E[f_k]$ which in its turn depends on the latency, we need to simultaneously solve the rate equations for $E[f_k]$ and $E[c_k]$ taking into account all the interleaved stochastic processes affecting them. This is in contrast to the model without delays [73] where we could solve first for the $E[f_k]$'s and input these values into the equation for $E[c_k]$.

Probability of failed fingers, f_k

As in our previous analysis, we only consider dead fingers and not those which are alive but wrong. Unlike members of the successor list, *alive* fingers even if outdated (which in Chord, means that instead of pointing to the first successor of an interval, sometimes the finger may be pointing to the second or third node) always bring a query closer to the destination. Hence they do not substantially affect the lookup length though they could affect the consistency. As mentioned earlier, we make an optimisation in the correction of the earlier fingers. All initial fingers which are also the successor are corrected at the same time as the successor. In addition, in order to artificially hold the ring together, the simulator provides a node with a successor every time the successor fails. This new successor also replaces the finger table entry for those fingers which pointed to the old successor. The remaining fingers are selected randomly, every time a finger stabilisation is scheduled. Let $f_k(r, \alpha, t)$ denote the instantaneous fraction of nodes whose kth finger points to a failed node and $F_k(r, \alpha, t)$ denote the respective number. For notational simplicity, we write these as simply F_k and f_k. These are random variables whose value fluctuates in time. They have however a well defined average in steady state $E[f_k]$ which we estimate here by keeping track of the gain and loss terms for F_k as a result of a join, failure or stabilization events. These are listed in table **??**.

The first term, c_1, is the probability for a node's kth finger to fail during a small time period Δt (probability $\lambda_f N \Delta t$) in the case that it was alive in the first place (probability $1 - f_k$). In this case the number of nodes with failed kth pointers increases by 1. Alternatively, we can think of this term as the increase in the number of failed kth finger pointers, every time a node fails in the system (without considering the failed nodes own pointer, which is done in term c_3).

The second term is the probability that an answer to a stabilization query is successfully returned to a sender with a failed kth pointer. In this case, the sender corrects the finger and hence the number of nodes with failed kth pointers decreases by 1. This probability is a product of a node having a dead kth finger (f_k) times the probability that an answer to a lookup query for the kth finger, is received. The latter term is in turn, a product of three factors. The factor $(1 - \alpha)\lambda_s N \Delta t$

$F_k(t + \Delta t)$	Probability of Occurrence
$= F_k(t) + 1$	$c_1 = (1 - f_k)(\lambda_f N \Delta t)$
$= F_k(t) - 1$	$c_2 = (1 - \alpha) \left\langle \frac{1}{N_d} \right\rangle f_k(\lambda_s N p \Delta t)$
$= F_k(t) - 1$	$c_3 = f_k(\lambda_f N \Delta t)$
$= F_k(t) + 1$	$c_4 = f_k(\lambda_j \Delta t)$
$= F_k(t)$	$1 - (c_1 + c_2 + c_3 + c_4)$

Table 7.1: The relevant gain and loss terms for F_k– the number of nodes whose kth fingers are pointing to a failed node for $k > 1$.

is the probability that a finger query is sent out. Since we are interested only in finger queries sent out for the kth finger, the factor $E[\frac{1}{N_d}]$ accounts for the fact that the protocol picks any finger equally likely for stabilisation, out of all fingers N_d which are not also the first successor. The average is carried out over all the nodes in the system. The value for $E[N_d]$ is close to $\log_2 N$, as expected from elementary considerations. However we can also estimate this more accurately by using the internode-interval distribution which gives the probability that a node has an empty interval of size exactly equal to x in front of it. This is simply $\rho^x(1-\rho)$ with $\rho = 1 - N/\mathcal{K}$ [73]. Using this it is easy to get an accurate estimate of $E[\frac{1}{N_d}]$ as shown in Appendix .3. The factor p in c_2 is a measure of the probability that the answer to a stabilisation query is actually used to correct a finger. If the sender of the query is alive when the query is answered (as is always the case without delays), then the dead finger is replaced. Hence if we set $p = 1$, we get an expression for the f_k's without delays (or with delays which are negligible with respect to node lifetimes). However with network-induced delays, since lookups can no longer be considered instantaneous, the sender could fail before receiving a reply. In this case, some fraction of lookup queries, $1 - p$ in our notation, are discarded even after completion. Hence p can have a value different from 1. We can estimate p by relating the lookup process to a series of M/M/∞ queues where the 'customers' in the queue are the lookup messages sent for the kth finger and the number of queues in series are the number of hops needed to lookup this finger. This is done in Appendix .2. Since the number of hops and hence the network-induced delay, depends on which finger is looked up, the value of p decreases as k increases. The terms c_3 and c_4 denote respectively the probability that a node with a failed kth pointer fails (hence decreasing F_k by 1) and that a node joins and copies a dead finger entry from its successor (as per the join protocol specified). Table **??** gives us a rate equation for the quantity $E[f_k]$. For ease of notation we represent $E[f_k]$ by f_k and $E[c_k]$ by c_k in all that follows. In steady state, we get

$$(1 - f_k) + f_k = f_k \tilde{r} p + f_k \tag{7.1}$$

where $\tilde{r} = (1-\alpha)r\left\langle \frac{1}{N_d} \right\rangle$. Hence

$$f_k = \frac{1}{1+p\tilde{r}} \tag{7.2}$$

Note that the form of $f_k = \frac{1}{1+\tilde{r}}$ for $p = 1$ is very intuitive if we consider each user as an alternating renewal process [123] that is ON when the user (and hence the kth finger that points to this user) is alive (with an expected lifetime $1/\lambda_f$) and OFF when the user leaves and a stabilisation has not yet been done by the owner of the affected finger (with an expected time interval $1/\tilde{\lambda_s}$).

Without delays, an answer is expected (and the finger corrected) as soon as a query is sent. With delays, this is not the case. The time interval that a dead finger lasts on average in a nodes routing table includes the wait for a stabilistion to be scheduled *and* then a delay of the order of c_k/λ_d for the answer to be received where c_k is the average number of hops including timeouts taken by a lookup for the kth finger. This translates to an effectively *s*lower stabilisation rate $p\tilde{r}$ with $p < 1$.

Using the approximation for the value of p derived in the appendix, we get

$$f_k = \frac{1}{1+\tilde{r}(1+1/d)^{-c_k}} \tag{7.3}$$

The above expression is exact in the case that the distribution of the latency of the kth finger is peaked around its average value (see Appendix .2). If this is not the case, expression $p = (1+1/d)^{-c_k}$ is an underestimate by Hölder's inequality. From figure 7.1, we see that the theoretical estimate does indeed underestimate the value obtained via simulations for the values of r, d and N that we have looked at.

For large delays or small d, $p \to 0$, since the sender of the query is almost always dead by the time the query is answered. This gives $f_k \sim 1$, which implies that the kth finger and all fingers larger than k are dead (and hence unusable) at all times. As we will see this actually happens in Chord depending on the values of the parameters N,d and r, due to non-linear effects that arise because of the coupling of equation 7.3 with the lookup equation developed below. Another outcome of this non-linear coupling between c_k and f_k is that the worst-case lookup time no longer scales as $\sim log(N)$, as soon as $d \neq 0$. We delve into this deeper in section 7.7.

Latencies and average number of hops, c_k

Equation 7.3 gives us the f_k's in terms of the $c_k's$. However as shown in our earlier analysis [73], the c_k's themselves also depend on the f_k's. For self-completeness we present this calculation again here.

To calculate the c_k's, the average number of hops required in looking up finger k, we need to understand how the Chord protocol executes a lookup for a key a distance t away from a given node who originates the query. Let c_t denote the expected cost (number of hops including time outs) for a given node to reach some

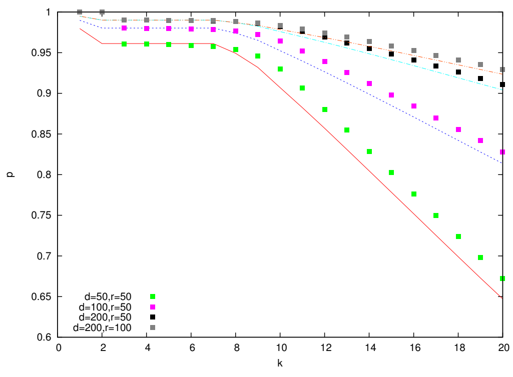


Figure 7.1: Comparison between theory (line) and simulation (dots) for different parameter values for the probability that a lookup for a kth finger is received by the sender while it is alive.

target key which is t keys away from it (which means reaching the first successor of this key). The expected cost for reaching a general distance t, in accordance with the Chord protocol, may be written recursively in terms of the expected cost of reaching a target closer than t. To do so, let us define ξ to be the *s*tart of the finger (say the kth) that most closely precedes t. Hence $\xi = 2^{k-1} + n$ and $t = \xi + m$, *i.e.* there are m keys between the sought target t and the start of the closest preceding finger. We can write a recursion relation for $c_{\xi+m}$ as follows:

$$\begin{aligned}
c_{\xi+m} &= c_{\xi}\left[1 - a(m)\right] \\
&+ (1 - f_k)a(m)\left[1 + \sum_{i=0}^{m-1} bc(i,m)c_{m-i}\right] \\
&+ f_k a(m)\Bigg[1 + \sum_{i=1}^{k-1} h_k(i) \\
&\sum_{l=0}^{\xi/2^i-1} bc(l,\xi/2^i)(1 + (i-1) + c_{\xi_i-l+m}) + O(h_k(k))\Bigg]
\end{aligned} \tag{7.4}$$

where $\xi_i \equiv \sum_{m=1,i} \xi/2^m$ and $h_k(i)$ is the probability that a node is forced to use its $(k-i)$th finger owing to the death of its kth finger. The coefficient $a(x) = 1 - \rho^x$ denotes the probability that there is at least one node present in an interval of length x and $b(x) = \rho^x(1-\rho)$ is the probability that there is an empty interval of exactly x positions after which there is a node present. $bc(i,x)a(x)$ is the conditional probability that the first node in an interval of length x is found i positions away, given that there is at least one node in this interval. All of these are elementary consequences of the geometric distribution of empty intervals on the ring. We could also consider corrections to these coefficients due to churn (a node may not know that it has a successor in the interval of interest), however these are very small in general and in our case strictly $= 0$ because of the optimisation we implement to keep the ring connected under high churn.

We now explain the equation briefly term by term. The first term accounts for the eventuality that there is no node intervening between ξ and $\xi + m$ (occurs with probability $1 - a(m)$). In this case, the cost of looking up $\xi + m$ is the same as the cost for looking up ξ. The second term accounts for the situation when a node does intervene in between (with probability $a(m)$), and this node is alive (with probability $1 - f_k$). Then the query is passed on to this node. The cost for this is an increase of 1 to register the increase in the number of hops and the remaining cost of looking up the distance between the new node and t. The third term accounts for the case when the intervening node is dead (with probability f_k). Then the cost increases by 1 (for a timeout) and the query is passed on to an alternative lower finger that most closely precedes the target. Let the $(k-i)$th finger (for some i, $1 \leq i \leq k-1$) be such a finger. This happens with probability $h_k(i)$ *i.e.*, the probability that the lookup is passed back to the $(k-i)$th finger either because the intervening fingers are dead or share the same finger table entry as the kth finger is denoted by $h_k(i)$. The start of the $(k-i)$th finger is at $\xi/2^i$ and the distance between $\xi/2^i$ and ξ is equal to $\sum_{m=1,i} \xi/2^m$ which we denote by ξ_i. Therefore, the distance from the $s$tart of the $(k-i)$th to the target is equal to $\xi_i + m$. However, note that the node which first precedes the start of finger $k-i$ could be l keys away (with probability $bc(l, \xi/2^i)$, for some l, $0 \leq l < \xi/2^i$). Therefore, after making one hop to the node of finger $k-i$, the remaining distance to the target is $\xi_i + m - l$. The increase in cost for this operation is $1 + (i-1)$; the 1 indicates the cost of taking up the query again by the node of finger $k-i$, and the $i-1$ indicates the cost for trying and discarding each of the $i-1$ intervening fingers. The probability $h_k(i)$ is easy to compute given the distribution of the nodes and the f_k's computed in the previous section.

$$\begin{aligned} h_k(i) =& a(\xi/2^i)(1 - f_{k-i}) \times \Pi_{s=1,i-1}(1 - a(\xi/2^s) + \\ & + a(\xi/2^s) f_{k-s}), i < k \\ h_k(k) =& \Pi_{s=1,k-1}(1 - a(\xi/2^s) + a(\xi/2^s) f_{k-s}) \end{aligned} \tag{7.5}$$

In equation 7.5 we account for all the reasons that a node may have to use its $(k-i)$th finger instead of its kth finger. This could happen because the intervening fingers were either dead or not distinct. The probabilities $h_k(i)$ satisfy the constraint $\sum_{i=1}^{k} h_k(i) = 1$ since clearly, either a node uses any one of its fingers or it doesn't. This latter probability is $h_k(k)$, that is the probability that a node cannot use any earlier entry in its finger table. In this case, n proceeds to its successor list. The query is now passed on to the first alive successor and the new cost is a function of the distance of this node from the target t. We indicate this case by the last term in Eq. 4 which is $O(h_k(k))$. This can again be computed from the inter-node distribution. However in practice, the probability for this is extremely small except for targets very close to n. Hence this does not significantly affect the value of general lookups and we ignore it in our analysis.

Equation 7.4 gives the expected cost as per the Chord greedy lookup protocol, for looking up a key *a*ny distance away from node n. In earlier work, we solved this equation numerically for *a*ll values of t, and averaged over all the values c_t to get an estimate of the expected cost of a general lookup. In this paper we are interested in solving equation 7.4 for each finger ($t = 2^i$), consistently with the equations for the f_k's (equation 7.3). Note that $t = 2^i$ does *n*ot imply that $m = 0$ in Equation 7.4, rather $m = 2^{i-1}$ since it is the distance from the start of the closest *p*receding finger that appears in the equation.

To solve Equations 7.3 and 7.4 simultaneously, we first set the f_k's for the earlier fingers ($\log \mathcal{K} - E[N_d]$ of them) to 0 as a result of the optimisation we make. For the remaining fingers, we start by setting f_k to some initial value. This is input into equation 7.4 to obtain a value of c_k. This value of c_k is in its turn used to calculate f_k using equation 7.3 and the whole process is repeated till the expressions for f_k and c_k converge. Note that the cost c_k for a finger k depends only on the costs of the earlier fingers (which have lower values of k). Hence Equation 7.4 may be solved finger by finger to obtain both f_k and c_k for all of the $E[N_d]$ distinct fingers.

7.5 Simulations

In order to validate the findings of our analysis we use a discrete event simulation of the version of the chord protocol presented in [73]. The simulation is driven by four independent Poisson processes for node joins, node failures, stabilisation events and message delivery with rates λ_j, λ_f, λ_s and λ_d respectively.

A join event is scheduled at startup and when executed, a node joins the system. At the same time, a new join event is scheduled an exponential time later according to the given rate. Every time a node joins, a fail event and a stabilisation event are scheduled locally for the given node. When the fail event is executed the node is (non-gracefully) removed from the system and when a stabilisation event occurs, the node executes the event and schedules a new event an exponentially distributed time later.

When a node tries to communicate with other nodes (either by sending a ping or a message) it generates a transmission event which is executed by the system an exponentially distributed (with parameter λ_d) time later. A lookup that involves 10 hops, say, which include timeouts when encountering departed nodes en-route, involves 10 such transmission events. In the case of a time-out, the message is discarded and the sender notified, with the same delay as for a successful hop. This network model then implies the presence of a failure detector operating on the same time scales as the network delay time. That is, we assume that there is no additional time elapsing between the moment the ping request arrives at the target and the sender being notified. In practice this means that there is no possibility of a ping failing when the target node is actually alive (false suspicion). Additionally, since nodes generate a transmission event as soon as they receive a message to forward, lookups are always completed, though the presence of dead nodes en-route can delay their completion.

The simulator is initiated with a perfect chord network with exactly N nodes. The simulation is then run for a warm-up period of $400N$ join/fail events until steady state is reached. This state is then sampled during an additional time of $400N$ and all measured quantities are averaged over this time period.

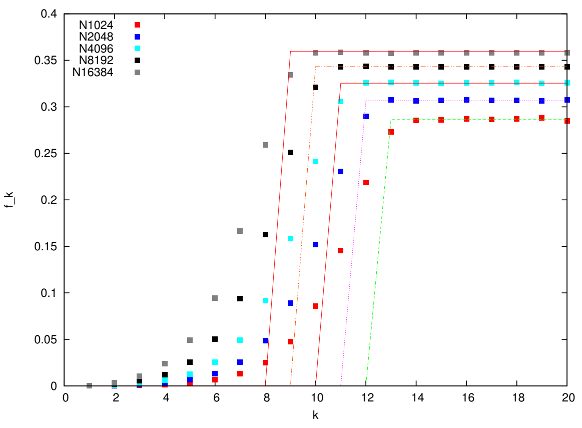
Since our main interest is in predicting the latency required for a lookup rather than stability properties of the ring itself, such as disconnection probabilities, we artificially hold the ring together as mentioned earlier.

7.6 Results

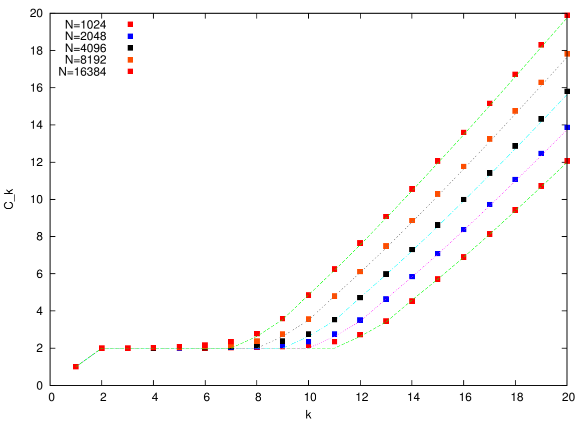
In this section we compare our theoretical prediction for the f_k's and c_k's (equation 7.3 and 7.4) against the steady state values of the same quantities from the simulator. In each figure, the left plot shows the steady state fraction of dead kth fingers, f_k, as a function of finger index $k = 1..20$. The right plot shows c_k, the lookup latency in number of network hops including the number of ping timeouts used to detect dead nodes en route.
Since we are interested in examining the performance range of the system we set the parameters to somewhat extreme values. In live implementations of large distributed systems (BitTorrent for instance [108]) the stabilisation rate of routing table entries is done on a timescale of minutes or more. Further, the session time for nodes in live DHT systems [103] can in many cases be as low as a minute. Similarly the round trip time for a message across the Internet can in many cases be delayed tens of seconds or more [30]. These facts let us conclude that it is not unreasonable to assume that λ_d, λ_s have the same order of magnitude as the session time λ_f. In our simulations, we use values of r ranging from 50 upto 1000 and values of d ranging from 50 to 200. As mentioned earlier, we only consider $r \geq d$ so that there is a steady state value of unanswered queries in the system.

As a comparison we first solve equation 7.3 and 7.4 without network delays ($d = \infty$) and high churn rate $r = 50$. Figure 7.2 shows the f_k's and c_k's for system sizes ranging from $N = 2^{10}$ to $N = 2^{14}$ nodes. We see that the theoretical predictions match the simulation results very well for all system sizes.

Next we study the effect of large delays. We set $d = 100$ while keeping the churn rate at $r = 50$. This value of d implies that the network-induced delay between any pair of nodes is on average a 100th of the average node lifetime. We see that even in this case we are able to predict the f_k's and c_k's with high precision. In contrast to the case without delays, the f_k's increase with finger index k. This is expected, as explained in section 7.4. It is interesting to note that the $c_k's$ at this value of delay are only very modestly over the value of the c_k's in Figure 7.2.



(a)



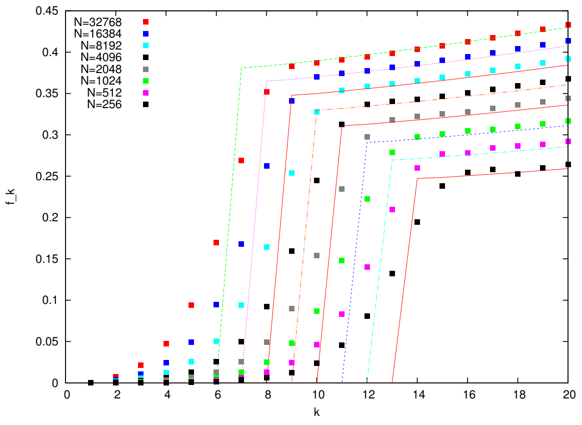
(b)

Figure 7.2: Simulation (points) and theory (lines) with churn rate $r = 50$ and no delays.

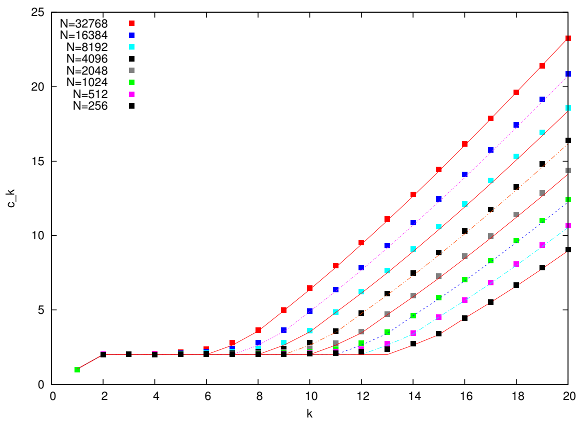
This is even more obvious in Figure 7.4 where we study the effect of varying d while keeping the number of nodes and churn rate constant. The zero-delay (or $d = \infty$) case is also plotted for comparison. As the delay increases (or d decreases), for a fixed k, both c_k and f_k increase, but the increase in c_k isn't large for these parameter values.

The effect of varying churn rate while keeping $N = 2^{14}$ and $d = 200$ is shown in figure 7.5.

From a first look at these figures it appears as if churn plays a larger role than the value of d. However note the large effect on both f_k and c_k when d is decreased



(a)

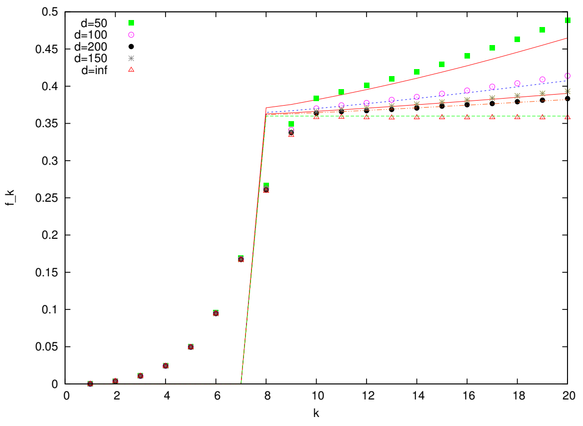


(b)

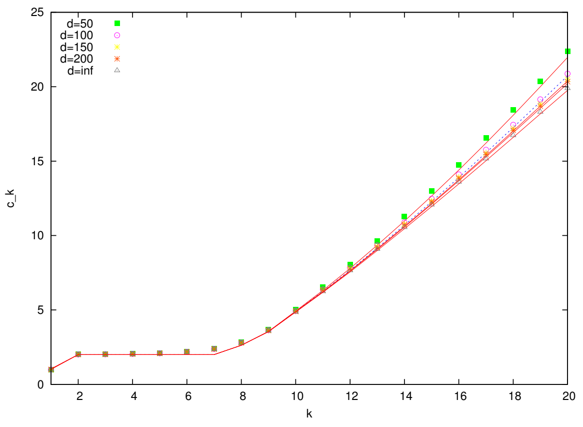
Figure 7.3: Simulation (points) and theoretical (lines) results for various system sizes and $d = 100$ and $r = 50$

from 100 to 50 as opposed to when it is decreased from 150 to 100 in Fig. 7.4. Such a behaviour seems to indicate that non-linear effects come into play at large delays. Indeed, as we will see in section 7.7, such effects do arise in Chord and are crucial to understanding parameter ranges appropriate for large systems.

We note that we are able to predict the outcome from the simulator with very high accuracy in all cases. However, as Fig. 7.4 shows, the quality of the match decreases at $d = 50, r = 50$ for the f_k's. This is presumably due to the fact that the approximation for the value of p deteriorates as d decreases as also indicated by Fig. 7.1.



(a)



(b)

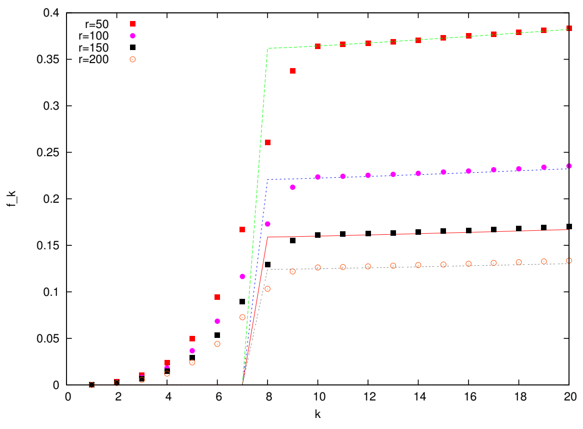
Figure 7.4: Simulation (points) and theory (lines) for $N = 2^{14}$ and $r = 50$ for different d.

7.7 Extensions

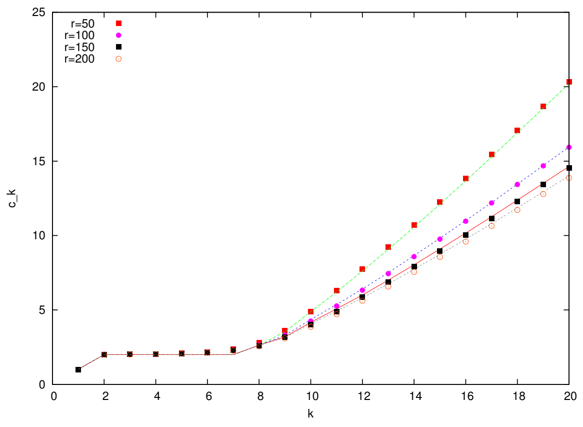
In this section we briefly describe how the model can be extended to analyse proximity-aware neighbour selection or routing policies as well as some interesting conclusions we come to on worst-case behaviour for large N.

Proximity neighbour selection (PNS)

In the model studied so far, we have assumed that the delays between any pair of nodes in the overlay is given by the same exponential distribution. In practice



(a)



(b)

Figure 7.5: Simulation (points) and theory (lines) for $N = 2^{14}$ and $d = 200$ for different r.

however, this is definitely not the case. We consider therefore a generalization of the above scenario in which there are two kinds of links present in the network, those that are fast and those that are slow. Such a model is for instance, applicable to scenarios where the network displays natural partitions such as multiple LANs in a WAN or cities on a regional back-bone network. We model the transition rates between pairs of nodes with two rates : λ_{d1} for *fast* links and λ_{d2} for *slow* links with $\lambda_{d1} > \lambda_{d2}$. Hence we now have *two* ratios $d_1 = \lambda_{d1}/\lambda_f$ and $d_2 = \lambda_{d2}/\lambda_f$ which characterize the system. Further we assume that there is a fixed time-independent fraction of links z which are slow and $1 - z$ which are fast.

Under the PNS scheme, every node tries to have fingers which are fast links. This could imply that a node has *no* fingers in an interval. But this is exponentially unlikely for the larger fingers.

We consider the following procedure for a node to populate its finger table. To get a contact in section $2^{i-1} \leq m < 2^i$ (its i^{th} finger), the node initiates a lookup for id 2^{i-1} in the usual way. With a probability $1-z$ the first successor of this id is a fast link in which case its kept as the i^{th} finger. With a probability z however, its a slow link. In this case, the node pings all the successors of the target node consecutively (we assume that for every lookup for a finger, the target node also sends back its full successor list) until it finds a fast link. This procedure may be considered a variant of the PNS(K) heuristic considered in [53], for example.

For greedy lookups done now with the above PNS optimisation, we need to recalculate both f_k and c_k. For the former, the term in equation 7.3 that we need to recompute is p - the probability that the answer to a finger query is actually used to correct a finger. Under the PNS scheme, not only do lookups take time, but if the lookup gives a bad answer then the subsequent pings take time as well. p now is the probability that the node that sent the query is still alive after this whole process.

The probability that the combined lookup process takes a time t is

$$f_T(t) = \int_0^t f(t')g(t-t')dt' \tag{7.6}$$

where the functions $f(t) = \frac{\lambda_{d1}^{c+1}}{\Gamma(c+1)} t^c e^{-\lambda_{d1} t}$ and $g(t) = \frac{\lambda_d^n}{\Gamma(n)} t^{n-1} e^{-\lambda_{d2} t}$ are the same as derived in appendix .1 and n is the number of pings that need to be done at the end of the lookup in order to find a fast link. The $c+1$ in the function $f(t)$ appears because the number of fast links include the c lookup hops as well as the last ping in the pinging process.

The derivation of p follows along the same lines as for the homogenous case and is briefly outlined in Appendix .2. To get the expected value of p, now both c and n have to be averaged over. While the distribution of c is $Q(c)$ as before, the distribution of n is simply the geometric distribution $z^n(1-z)$.

As before we need to solve the equations for the f_k's simultaneously with the equations for the c_k's. However we need to rewrite the lookup equation 7.4 slightly so that we account for the possibility that the first node in the interval is the entry in the routing table for finger k *o*nly if it is also a fast link.

The lookup equation has the same logical structure as Eq. 7.4 except for one difference. The coefficient $a(m)$ - the probability of having at least one node in an interval of size m, is replaced by $y(m)$, the probability of having a *finger* in an interval m, since we are no longer guaranteed that the first node in an interval is the finger of choice.

$y(m)$ is easily calculated recursively in the following manner. Let $y^{'}(m)$ be the probability of *n*ot having a finger in an interval of size m. Clearly $y^{'}(m) = y^{'}(m-1)(\rho + (1-\rho)z)$, that is, the probability of not having a finger in an interval of size m is the same as the probability of not having a finger in an interval of size $m-1$, times the probability that either there exists no node at the m^{th} spot or there is one but with a slow link. Solving the recursion, we get $y(m) = 1-(\rho+(1-\rho)z)^m$. When $z = 0$ as in the case with homogenous links, $y(m)$ is the same as $a(m)$.

With this, we can write the lookup equation in the PNS case -

$$\begin{aligned}
&c_{\xi+m} = c_\xi\,[1-y(m)]\,(1-z)\\
&+[1-y(m)]\,z\left[\sum_{i=1}^{k-1} h_k(i)\right.\\
&\left.\sum_{l=0}^{\xi/2^i-1} bd(l,\xi/2^i)(1+(i-1)+c_{\xi_i-l+m})+O(h_k(k))\right]\\
&+(1-f_k)y(m)\left[1+\sum_{i=0}^{m-1} bd(i,m)c_{m-i}\right]\\
&+f_k y(m)\left[1+\sum_{i=1}^{k-1} h_k(i)\right.\\
&\left.\sum_{l=0}^{\xi/2^i-1} bd(l,\xi/2^i)(1+(i-1)+c_{\xi_i-l+m})+O(h_k(k))\right]
\end{aligned} \tag{7.7}$$

The logical structure is almost the same as before. The first term calculates the probability for the case when there is no finger in the interval m but the first successor of m is the entry in the finger table (which happens with probability $1-z$). In this case, the cost of looking up ξ and $\xi+m$ are the same since they are owned by the same node. The second term accounts for the possibility that neither is there a finger in the interval, nor is the first successor of the interval m the entry in the finger table. In this case, ξ and $\xi+m$ are not owned by the same node and hence the cost is explicitly written as a sum which includes back tracking to a previous finger. The third and fourth terms are very similar to those in equation 7.4 and account for the cases when there is a finger which is alive or there is a finger which points to a departed node. Note that the back-tracking terms are the same in the second and fourth terms except for an additive factor of 1 which is present in the fourth term because of the cost of a timeout. The function h_k is the same as before with the $a(m)$'s replaced by $y(m)$'s and the functions bd's are the counterpart of the bc's which appear in Eq. 7.4 and are given in terms of the $y(m)$'s as

$$bd(i,m) = \frac{(1-y(i-1))(1-\rho)(1-z)}{y(m)}$$

Proximity Route Selection

We briefly demonstrate here how we can also analyse proximity route selection. In this case we do not need to recompute the probability of fingers being dead since this is the same as equation as in Eq. 7.3. But we need to consider a PRS scheme for routing. There clearly exist many such schemes [101], but we consider the simplest based on pure proximity. In this case, a node, uses a finger for routing *o*nly if its a good link, no matter how far back it has to hop from the target. Since the manner in which fingers are chosen is the same as in the case without proximity, The lookup equation is the same as Eq. 7.4 except that the h_k's are now a little different, to account for the fact that a jump back to an earlier finger can be made even if the finger was alive but a bad link.

The expression for the h_k's is hence:

$$\begin{aligned} h_k(i) =& a(\xi/2^i)(1-f_{k-i})(1-z) \\ & \times \Pi_{s=1,i-1}(1-a(\xi/2^s)+a(\xi/2^s)f_{k-s}+ \\ & +(1-a(\xi/2^s))(1-f_{k-s})z), i<k \\ h_k(k) =& \Pi_{s=1,k-1}(1-a(\xi/2^s)+a(\xi/2^s)f_{k-s}+ \\ & +(1-a(\xi/2^s))(1-f_{k-s})z) \end{aligned} \tag{7.8}$$

We could also consider schemes where we optimise how far we hop back to find a good connection versus how much this increases the lookup latency.

Iterative routing

Rather than using sequential routing where each message is relayed by the nodes in the message path it is possible to implement the so called *iterative routing.* In iterative routing each node on the path returns the address of the next node to the originator of the message. Our model can easily be adapted to such a scheme. The only effect is that each successful hop on the path includes one more transmission back to the sender. Hence in equation 7.4 in the second term, instead of adding 1 to the cost we need to add 2. In the third term, the successful relay of a message is done when the lookup is sent back to an alive finger. The change we need to do to this term is the to change the cost $1+(i-1)$ to $2+(i-1)$. The f_k equation is not affected.

Worst-case predictions for large N

As we have seen, our analysis gives numbers very close to simulations for all reasonable values of the parameters that we have studied and in addition is easily extendable to include various proximity-aware schemes. We would however also like to understand if the theory we have developed helps us make predictions for large systems. Actual implementations of Chord sometimes use a 160 bit identifier space [109] ($\mathcal{K}=2^{160}$) and one could imagine a situation with many millions or even billions of participating nodes. $\mathcal{K}$ by itself is not an important parameter. Though it determines the number of fingers (or degree) of each node, only $\sim \log_2 N$ of the fingers differ from the first successor (as mentioned earlier) and the values of c_k for these fingers depend only on the parameters N, d and r. In other words, the last 15 fingers for $N = 2^{15}$ and $\mathcal{K} = 2^{160}$ have the same values of c_k as the last 15 fingers for the same N and $\mathcal{K} = 2^{20}$, all other parameters being the same. Increasing N however, or changing d and/or r does change the c_k's and f_k's more non-trivially as shown in the Figures. We would hence like to understand if there are any combinations of parameter values N, r and d for which the theory predicts a problem such as lookups taking an arbitrarily long time or permanently dead routing-table entries.

To make such an estimate we need a closed form expression for the expected cost c_k for finger k in terms of f_k. Such a closed form expression along with Eq. 7.3 would then give us the possibility to eliminate f_k and get a self-consistent equation for c_k, making it easy to predict numbers for arbitrarily large systems. Equation 7.4, though accurate in its predictions has an unwieldy form in this regard, and its difficult to derive a closed form expression from it. To help us round this point however, we can use a result in the paper by Aspnes *e*t al [10]. As mentioned in Section 7.3, bounds are derived here for the lookup time taken by a generic greedy routing algorithm in the case when nodes are embedded along a one-dimensional real line. Specifically, for the case when a fraction p of long-distance links are present and the average degree of nodes scales as $\sim \log(N)$, they find an upper bound for lookup time which goes as $\sim B/p$ where B is the corresponding upper bound with no failures. If we translate this into our language, then for each k, p corresponds to $1 - f_k$, and we can use the value c_{k0} for B, where c_{k0} is the lookup latency without churn. This latter quantity is easily obtained from Equation 7.4 by setting all the $f_k's$ to 0, when the equation reduces to the simple recursion

$$C_{i+1} = \rho C_i + (1-\rho) + (1-\rho) C_{i+1+\xi(i+1)} \qquad (7.9)$$

$\xi(i+1)$ is the value of the start of the finger most closely preceding the key $i+1$. We have verified earlier [73] that equation 7.9 gives a very accurate description of the Chord greedy lookup protocol without churn. It is a consequence of the above equation that the cost for looking up finger k for a node at position 0 depends only on the number of nodes N_k between 0 and 2^{k-1} and scales as $\sim \log(N_k)$. If we take the prediction of Equation 7.9 as our estimate of c_{k0} and use $p = 1 - f_k$, we get

$$c_k = \frac{c_{k0}}{1 - f_k} \qquad (7.10)$$

Note that only $f_{k'}$ for $k' < k$ appear in the expression for c_k according to Equation 7.4. Hence replacing p by $1 - f_k$ is equivalent to taking the smallest possible value of p for predicting the cost c_k which is consistent with Equation 7.10 remaining an overestimate.

As we see from Figure 7.6, Equation 7.10 does indeed seem to overestimate the data from simulations as it would if it was a true bound. Here we have taken our theoretical estimate of f_k (Equation 7.3) and the value of c_{k0} generated from Eq. 7.9 to obtain an estimate of c_K which we compare with simulations.

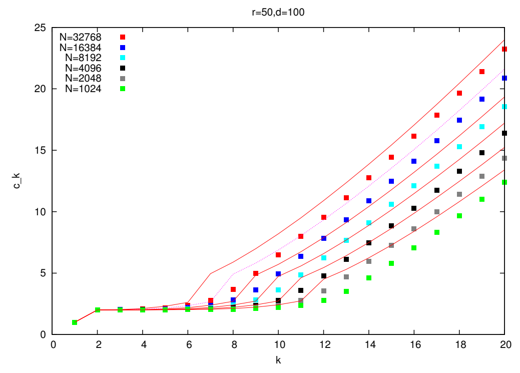


Figure 7.6: Comparing the expression $c_{k0}/(1-f_k)$ (lines) with simulation data (points) for different system sizes, churn value $r = 50$ and a network delay $d = 100$. Though the expression $c_{k0}/(1-f_k)$ is expected to be an over-estimate as explained in the text, it seems a good estimate even for the average value.

Given this motivation for using Equation 7.10, we can investigate its implications by eliminating f_k using Eq. 7.3. By this means we get a self-consistent equation for c_k:

$$(c_k - c_{k0})\tilde{r}e^{-ac_k} = c_{k0} \tag{7.11}$$

where $a = \log(1 + 1/d)$. Note that the three parameters that we can vary independently in our simulations N, d and r, also appear here in the guise of c_{k0}, a and $\tilde{r}$ respectively. Equation 7.11 is a common form of a transcendental equation and its solution may be written formally in terms of the Lambert W function [29] as

$$c_k = c_{k0} - \frac{W(-ac_{k0}e^{ac_{k0}}/\tilde{r})}{a}. \tag{7.12}$$

From analysing equation 7.11 or equivalently using the property that $W(x)$ is real and negative only if x is in the range $-1/e < x < 0$, we find that for any value of $\tilde{r}$ and c_{k0}, d has to be larger than (or a has to be smaller than) a certain value for solutions to exist. This boundary value of $a \geq 0$ is given by the condition $x = -1/e$ which translates in our case to the transcendental equation

$$\frac{\tilde{r}e^{-ac_{k0}}}{c_{k0}} = ae \tag{7.13}$$

If a is smaller than this value then equation 7.11 has two valid roots and we take the smaller of these to be the value of c_k (since this is the value that reduces to the value of c_k without delays as $a \to 0$). If a is larger than this value, $f_k = 1$ for the given finger as well as all larger fingers. To take an example, let $\tilde{r} = 1.5$ (which is around the largest value of churn that we have considered in our simulation) in Equation 7.13. In this case, if $ac_{k0} \geq 0.4$ then $f_k \sim 1$ for some k (and clearly for all k larger than this value as well). This implies that if $d = 15$ or $a = 0.065$, we expect all fingers which have a $c_{k0} \geq 6$ to be dead all the time. For a system with 4 million nodes for example, this implies that the 16 largest fingers are dead all the time at these values of the parameters. Or in the case $N = 2^{30}$, this implies that the 24 largest fingers are dead all the time. Comparing the above predictions with the theoretical predictions obtained by solving Equations 7.3 and 7.4 simultaneously, show that the above estimates are surprisingly accurate. While these parameter values, at which a large number of routing table entries are permanently unusable, might be extreme, and the model of network delays over-simplifed, we feel it is still useful to have such an analysis, since it may set the stage for setting more rigorous bounds on such complex systems in the future. In addition the existence of such a regime in parameter space leads to a very important conclusion even in the range of parameters that the system does function, as we argue below.

From Equation 7.11 we can estimate the lookup latency for any system size of interest, for any value of d and r, if we know the corresponding lookup latency in the ideal case without delays or churn. For example, in the case with no network delays ($a = 0$), the lookup latency with churn c_k is $\sim (1 + \frac{1}{\tilde{r}})c_{k0}$. Since $c_{k0} \sim b_0 \log_2 N_k$, where N_k is the expected number of nodes in an interval of size $2^{k-1}(= 2^{k-1}(N/\mathcal{K}))$, this implies that without delays but with churn, the lookup latencies for the different fingers scale with system size just as they do without churn. However with delays, this is no longer the case, and c_k does *not* scale simply as $\log N_k$. To undertand how it does scale, we note that for $|x| < 1/e$ the Lambert W function may be expanded in a convergent series

$$W(x) = \Sigma_{n\geq 1} \frac{(-n)^{n-1}}{n!} x^n \tag{7.14}$$

For us $x = -ac_{k0}e^{ac_{k0}}/\tilde{r}$. If we keep terms only to first order in x, we get $W(-x) \sim -x$. Since $c_{k0} \sim b_0 \log_2(N_k)$ with $b_0 \sim 1$, this implies from equation 7.12 that

$$c_k \sim c_{k0}(1 + \frac{N_k^\gamma}{\tilde{r}}) \tag{7.15}$$

to first order, with $\gamma = ab_0/\log(2)$. If $a = 0$, no matter what the value of churn, the theory predicts that the $\log(N)$ scaling doesnt change. However, whenever the second term dominates, the scaling of c_k changes from $\log_2(N_k)$ to $N_k^\gamma \log_2(N_k)$. Equation 7.12 (or Eq. 7.15) hence predicts a very interesting interplay between churn, delay and system size. The effect of network delays may be enhanced or mitigated by either the churn value or the number of users in the network to result in lookups scaling with system size differently from the desired $\log N$.

Looking again at the figures in section **??**, for the value of churn and delay considered in Fig. 7.3, the second term in Eq 7.15 is only of the order of ~ 0.7

for the largest finger of the largest system size considered. Hence the $N^{\gamma}\log(N)$ scaling isnt yet visible. However the fact that Figure 7.4 indicates that doubling the value of the delay doesnt change either f_k or c_k in any linear manner, is consistent with our predictions.

7.8 Discussion and Conclusion

We have, in this paper, created a detailed model of the Chord DHT running on a network with node to node delays and node churn. While the analysis in this paper builds on earlier work [73], the new results here involve understanding how the routing performance, in terms of the probability for a node to have a live or dead kth finger as well as the number of hops required for a lookup, is affected by the interplay between node-churn, link delays and system size. We have compared our theoretical predictions with results from a discrete event simulator running the Chord protocol, and find that the theory matches the simulations with high accuracy for a wide range of parameters. We have also highlighted some possible extensions of the model which can be used to analyse different proximity-aware routing policies, hence providing an analytical framework for discussing the combined effect of churn and network delays in DHT's.

To our mind, the most interesting outcome of the analysis is the interplay between network delay and churn which results in parameter regimes where some fraction of the routing table entries of all the nodes in the system are permanently dead. Even in the case that we keep the parameters away from such values, our analysis predicts that the worst case lookup time in the presence of link delays grows faster than $\log(N)$, as $N^{\gamma}\log(N)/\tilde{r}$), where γ depends on link delays and $\tilde{r}$ is a measure of churn. This scaling kicks in anytime that either system size, link delays or churn values makes the term $N^{\gamma}/\tilde{r}$ comparable to 1 implying that evaluating the suitability of ring-based DHT's for large-scale applications is a complex task involving optimizing system size according to network conditions and churn, if performance guarantees are to hold. It would be interesting in this context to understand if utilizing a PNS scheme (as indicated in [31]), or making stabilisations reactive (which can also be done in the framework of this model [72]), could ameliorate the situation. Even in the case we have studied, it is of great interest to understand whether the change in scaling is only a worst-case result or could also be true in the average case (as indicated by Fig 7.6).

Our aim in this work has been to provide a model which can be used as a stepping stone towards gaining a further understanding of whether its possible to provide a fully distributed lookup service on the Internet, with billions of participating nodes. To this effect, the Chord protocol we analyse is entirely confined to the routing part of the application layer (storage and memory management not being specified), and the model of link-delays that we use is the most simplistic one possible. However our prediction that there could be potential problems even in this case, implies that other practical implementation issues that we have ignored, such as the presence of network address translators (NATs), network congestion or port conflicts etc, can only deteriorate working conditions even further.

7.9 Appendix

.1 Probability distribution for the time taken by c hops

Consider a lookup that takes c hops starting at the originator and ending at the target. Each of these c hops, which include timeouts for dead nodes, could be delayed by an amount of time t distributed exponentially with rate parameter λ_d. We can hence model this whole process as a series of c M/M/∞ queues, each with an exponential service time $g(t) = \lambda_d e^{-\lambda_d t}$. We would now like to estimate the probability $f(t)$ that a message is delivered in exactly a time t. The CDF for this quantity is the probability that the total transmission time $\leq t$:

$$F(t) = \int_0^t f(t') = 1 - \int_t^\infty f(t') \tag{16}$$

This probability is calculated by noting that the total transmission time is larger than t whenever less then c transmission events occur in that period. From our model of message delays, the probability that x events occur during a time t is given by a Poisson distribution with rate parameter λ_d. Thus, the probability of seeing no more then $c - 1$ events during time t is given by the sum:

$$= \sum_{x=0}^{c-1} (\lambda_d t)^x \frac{e^{-\lambda_d t}}{x!} \tag{17}$$

Hence:

$$F(t) = 1 - e^{-\lambda_d t} - \sum_{x=1}^{c-1} (\lambda_d t)^x \frac{e^{-\lambda_d t}}{x!} \tag{18}$$

This implies that:

$$\begin{aligned} f(t) =& \frac{dF(t)}{dt} = \lambda_d e^{-\lambda_d t} \\ & - \sum_{x=1}^{c-1} \frac{1}{x!} (x\lambda_d (\lambda_d t)^{x-1} e^{-\lambda_d t} - (\lambda_d t)^x \lambda_d e^{-\lambda_d t}) \Rightarrow \\ f(t) =& \lambda_d e^{-\lambda_d t} \left[1 - \sum_{x=1}^{c-1} (\frac{(\lambda_d t)^{x-1}}{(x-1)!} - \frac{(\lambda_d t)^x}{x!}) \right] \end{aligned} \tag{19}$$

All terms except the first and the last in the sum cancel:

$$\Rightarrow f(t) = \frac{\lambda_d^c}{\Gamma(c)} t^{c-1} e^{-\lambda_d t} \tag{20}$$

.2 Derivation of the value p

The value p denotes the probability that a lookup is actually returned to the originator. If however the time to complete the lookup is so long, that the originating node fails in the mean time, then the answer is discarded even though the lookup is completed.

The probability for a node to be alive upto at least a time t (given that it was alive at $t = 0$ when it originated a lookup) is $q(t) = e^{-\lambda_f t}$ and the probability density function for the time to complete a lookup is (see previous Appendix) $f(t) = \frac{\lambda_d^c}{(c-1)!} t^{c-1} e^{-\lambda_d t}$. Hence the probability of the originator of a query being alive when the lookup is completed is :

$$\begin{aligned} p(c) = \int_0^\infty f(t)q(t)dt = \frac{\lambda_d^c}{(c-1)!} \int_0^\infty t^{c-1} e^{-t(\lambda_d+\lambda_f)} dt = \\ \frac{\lambda_d^c}{(c-1)!} \frac{\Gamma(c)}{(\lambda_d+\lambda_f)^c} = \frac{1}{(1+\frac{1}{d})^c} \end{aligned} \tag{21}$$

The expected value of $p(c)$ is then given by:

$$E[p] = \int_0^\infty \frac{1}{(1+\frac{1}{d})^c} Q(c)dc = E[e^{-ac}] \tag{22}$$

Where $Q(c)$ is the probability that a lookup will require c transmissions and $a = \log(1 + 1/d)$. $E[p]$ has hence the form of the moment generating function of $Q(c)$.

An upper bound on $p(c)$ and hence also $E[p]$ might be obtained by replacing c by the corresponding value without delays (but with churn), since the value of the lookup with delays is always greater than the value without delays. We would however like to keep the dependence on $E[c]$ and assume to this effect that the principle contribution to $Q(c)$ comes from values of c around $E[c]$. This gives us the approximation

$$\hat{p} = (1 + \frac{1}{d})^{(-E[c])} \tag{23}$$

By the property of moment-generating functions and Hölder's inequality, this approximation is an underestimate for the true value Eq. 22. It nevertheless gives an excellent match with simulations as well as the correct limits at $d \to 0$ and $d \to \infty$.

If we assume a dichotomous delay distribution used in the PNS analysis in Section 7.7, then we can calculate the value p_{PNS} in a similar way. In this case Eq.21 becomes:

$$p(c,n) = \int_0^\infty f_T(t)q(t)dt \tag{24}$$

where $f_T(t)$ is obtained from Eq. 7.6, c is the latency of the lookup and n is the number of pings. p_{PNS} is obtained from averaging over both c and n and since the lookup and ping processes are independent of each other, the simultaneous probability of having c hops and n pings is simply the product $Q(c)R(n)$, where further $R(n) = z^n(1-z)$. The expected value of p_{PNS} is hence

$$E[p_{PNS}] = \int_0^\infty \int_0^\infty p(c,n)Q(c)R(n)dcdn \tag{25}$$

.3 Derivation of N_d

Let N_d be the number of fingers possessed by a node j which are *not* also the first successor. The ratio r/N_d then gives the rate at which j stabilises these fingers. Clearly, N_d is a random variable with a value which varies in time as well as varies from one node to the next. However, it has a well defined average value, when averaged over all nodes in the steady state. We can compute it in the following manner using the internode interval distribution described in the text.

If there is *no* node up to a key 2^{i-1} away from j, and at least one node in the interval $[2^{i-1}, 2^i - 1]$ then exactly i fingers of node j will point to this node which will also be j's first successor. The fingers which are not the first successor are then $N_d = \mathcal{M} - i$ where $\mathcal{M} = \log_2 \mathcal{K}$ is the total number of fingers.

Hence

$$\left\langle \frac{1}{N_d} \right\rangle = \sum_{i=1}^{M-1} (1-\rho)^{2^{i-1}-1}(1-\rho^{2^{i-1}})\frac{1}{\mathcal{M}-i} \tag{26}$$

The above expression gives an estimate quite close to the naive approximation $< N_d > \sim \ln(N)$.

Chapter 8

Modeling the Performance of Ring Based DHTs in the Presence of Network Address Translators

John Ardelius[1] and Boris Mejías[2],

[1] Swedish Institute of Computer Science (SICS), Sweden
john@sics.se

[2] Université catholique de Louvain, Belgium
boris.mejias@uclouvain.be

Abstract

Dealing with Network Address Translators (NATs) is a central problem in peer-to-peer applications on the Internet today. However, most analytical models of overlay networks assume the underlying network to be a complete graph, an assumption that might hold in evaluation environments such as PlanetLab but turns out to be simplistic in practice. In this work we introduce an analytical network model where a fraction of the communication links are unavailable due to NATs. We investigate how the topology induced the model affects the performance of ring based DHTs. We quantify two main performance issues induced by NATs namely large lookup increased break-up probability, and suggest how theses issues can be addressed. The model is evaluated using discrete based simulation for a range of parameters.

8.1 Introduction

Peer-to-peer systems are widely regarded as being more scalable and robust than systems with classical centralised client-server architecture. They provide no single point of failure or obvious bottlenecks and since peers are given the responsibility to maintain and recover the system in case of departure or failure they are also in best case self-stabilising.

However, many of these properties can only be guaranteed within certain strong assumptions, such as moderate node churn, transitive communication links, accurate failure detection and NAT transparency, among others. When these assumptions are not met, system performance and behaviour might become unstable.

In this work we are investigating the behaviour of a peer-to-peer system when we relax the assumption of a transitive underlying network. In general, a set of connections are said to be *non-transitive* if the fact that a node A can talk to node B, and B can talk to C does *not* imply that node A can talk to C. Non-transitivity directly influences a system by introducing false suspicions in failure detectors since a node cannot a-priori determine if a node has departed or is unable to communicate due to link failure.

In practice this study is motivated by the increasing presences of Network Address Translators (NATs) on today's Internet [32]. As many peer-to-peer protocols are designed with open networks in mind, NAT-traversal techniques are becoming a common tool in system design [104].

One of the most well studies peer-to-peer overlays, at least from a theoretic point of view, is the distributed hash table (DHT) Chord [109]. Chord and other ring based DHTs are especially sensitive to non-transitive networks since they rely on the fact that each participating node needs to communicate with the node succeeding it on the ring in order to perform maintenance. Even without considering NATs the authors of Chord [109], Kademlia [87], and OpenDHT [102], experienced the problems of non-transitive connectivity when running their networks on PlanetLab [1], where all participating peers have public IP address. Several patches to these problems have been proposed [46] but they only work with if the system has very small amount of non-transitive links, as in PlanetLab, where every node has a public IP address.

In this work, we construct an analytic model of the ring-based DHT, Chord, running on top of non-transitive network under node churn. Our aim is to quantify the impact a non-transitive underlay network to the performance of the overlay application. We evaluate the systems working range and examine the underlying mechanisms that causes its failure in terms of churn rate and presence of NATs. Our results indicate that it is possible to patch the Chord protocol to be robust and provide consistent lookups even in the absence of NAT-traversal protocols. Our main contributions are:

[1]PlanetLab: An open platform for developing, deploying, and accessing planetary-scale services. `http://www.planet-lab.org`

- Introduction of a new inconsistency measure, Q, for ring based DHT's. Using this metric we can quantify the amount of potential lookup inconsistency for a given parameter setting.

- Quantification of the load imbalance. We show that in the presence of NATs, nodes with open IP addresses receive unproportional amounts of traffic and maintenance work load.

- A novel investigation of an inherent limitation for the number of nodes that can join the DHT ring. Since each node needs to be able to communicate with its successor the available key range in which a node behind a NAT can join is limited.

- Evaluation of two modifications to the original Chord protocol, namely predecessor list routing and the introduction of a recovery list containing only peers with open IP addresses.

Throughout the paper we will use the words *node* and *peer* interchangeably. We will also use the term *NATed* or expressions such as *being behind a NAT* for a node behind a non-traversable NAT. It simply means that two nodes with this property are unable to communicate directly. Section 8.2 discusses related work, which justifies our design decisions for the evaluation model, which is discussed in Section 8.3. We present our analysis on lookup-consistency and resilience in Sections 8.4 and 8.5 respectively. The paper concludes by discussing some limitations on the behaviour of Chord and ring based DHTs in general.

8.2 Related Work

Understanding how peer-to-peer systems behave on the Internet has received a lot of attention in the recent years. The increase of NAT devices has posed a big challenge to system developers as well as those designing and simulating overlay networks. Existing studies are mostly related to systems providing file-sharing, voice over IP, video-streaming and video-on-demand. Such systems use overlay topologies different from Chord-like ring, or at most they integrate the ring as one of the components to provide a DHT. Therefore, they do not provide any insight regarding the influence of NAT on ring-based DHTs.

A deep study of Coolstreaming [80], a large peer-to-peer system for video-streaming, shows that at least 45% of their peers sit behind NAT devices. They are able to run the system despite NATs by relying on permanent servers logging successful communication to NATed peers, to be reused in new communication. In general, their architecture relies on servers out of the peer-to-peer network to keep the service running. A similar system, PPLive, that in addition to streaming provides video-on-demand. Their measurements on May 2008 [59] indicates 80% of peers behind NATs. The system also uses servers as loggers for NAT-traversal techniques, and the use of DHT is only as a component to help trackers with file distribution.

With respect to file-sharing, a study on the impact of NAT devices on BitTorrent [84] shows that NATed peers get an unfair participation. They have to contribute more to the system than what they get from it, mainly because they cannot connect to other peers behind NATs. It is the opposite for peers with public IP addresses because they can connect to many more nodes. It is shown that the more peers behind NATs, the more unfair the system is. According to [60], another result related to BitTorrent is that NAT devices are responsible for the poor performance of DHTs as “DNS” for torrents. Such conclusion is shared by apt-p2p [33] where peers behind NATs are not allowed to join the DHT. Apt-p2p is a real application for software distribution used by a small community within Debian/Ubuntu users. It uses a Kademlia-based DHT to locate peers hosting software packages [87]. Peers behind NATs, around 50% according to their measurements, can download and upload software, but they are not part of the DHT, because they break it. To appreciate the impact NAT devices are having on the Internet, apart from the more system specific measurements referenced above, we refer to the more complete quantitative measurements done in [32]. Taking geography into account, it is shown that one of the worst scenarios is France, where 93% of nodes are behind NATs. One of the best cases is Italy, with 77%. Another important measurement indicates that 62% of nodes have a time-out in communication of 2 minutes, which is too much for ring-based DHT protocols. Being aware of several NAT-traversal techniques, the problem is still far from being solved. We identify Nylon [65] as promising recent attempt to incorporate NATs in the system design. Nylon uses a reactive hole punching protocol to create paths of relay peers to set-up communication. In their work a combination of four kinds of NATs is considered and they are being able to traverse all of them in simulations and run the system with 90% of peers behind NATs. However, their approach does not consider a complete set of NAT types. The NATCracker [104] makes a classification of 27 types of NATs, where there is a certain amount of combinations which cannot be traversed, even with the techniques of Nylon.

8.3 Evaluation Model

Chord model

Chord [109] is a called distributed hash table (DHT) which provides a key-value mappings and a distributed way to receive the value for a specific key, a *lookup*. The keys belongs to the range $[0 : 2^K[$ ($K = 20$ in our case). Each participating node is responsible for a subset of this range and stores the values that those keys map to. It is important that this responsibility is strictly divided among the participating peers to avoid inconsistent lookups. In order to achieve this each node contains a pointer to the node responsible for the range succeeding its own, its *successor*.

Since the successor might leave the system at any point a set of succeeding nodes are stored in a *successor list* of some predefined length.

Lookups are performed by relying the message clockwise along the key range direction. When the lookup reaches the node preceding the key it will return the identifier of its successor as the owner of the key. In order to speed the process up some shortcuts are created known as *fingers*. The influence of NATs to the fingers is limited and are only discussed briefly in this work. Each nodes performs maintenance at asynchronous regular intervals. During maintenance the node pings the nodes on its successor list and removes those who have left the system. In order to update the list the node queries its successor for its list and appends the successors id.

NAT model

In order to study the effect of NATs we construct a model that reflects the connectivity quality of the network.

We consider two types of peer nodes: *open* peers and *NATed* peers. An open peer is a node with public IP address, or sitting behind a traversable-NAT, meaning that it can establish a direct link to any other node. A NATed peer is a node behind a NAT that cannot be traversed from another NATed peer, or that it is so costly to traverse, that it is not suitable for peer-to-peer protocols. Open peers can talk to NATed peers, but NATed peers cannot talk with each other. In the model, when joining the system, each node has a fixed probability p of being a NATed peer. The connectivity quality q of a network, defined as the fraction of available links will then be:

$$q = 1 - p^2 = 1 - c \tag{8.1}$$

Where c is the fraction of unavailable links in the system. Proof of equation 8.1 is straightforward and can be found in appendix [2].

We assume, without loss of generality, that all non-available communication links in the system are due to NATs. In practice we are well aware of the existence of several more or less reliable NAT-traversal protocols. [104] provides a very good overview and concludes that some NAT types, however, are still considered non-traversable (for instance random port-dependent ones) and the time delays they introduce might in many cases be unreasonable for structured overlay networks.

Churn model

We use an analytic model of Chord similar to the one analysed in [71] to study the influence of churn on the ring. Namely we model the Chord system as a M/M/∞ queue containing N (2^{12} in our case) nodes, with independent Poisson distributed join, leave and stabilisation events with rates λ_j, λ_f and λ_s respectively. The fail

[2]`http://www.info.ucl.ac.be/~bmc/apx-proof.pdf`

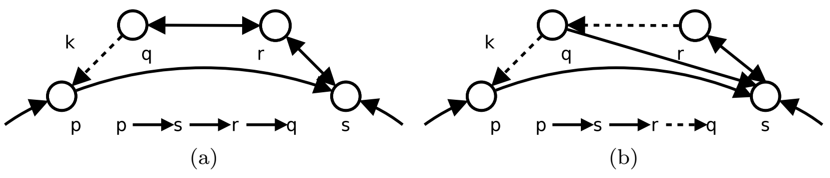


Figure 8.1: The creation of branches. Dotted lines indicates that the identifier of the peer is known but unable to communicate due to an intercepting NATs. 8.1a Peers p and q cannot communication which leaves peers p and r in a *branch* rooted at peer s. 8.1b In order to route messages to peers q and r, s has to maintain pointers to both of them in its *predecessor list* since path $s \rightarrow r \rightarrow q$ is not available.

event considers both graceful leaves and abrupt failures and the stabilisation is done continuously by all nodes in the system.

The amount of churn in the system is quantified by the average number of stabilisation rounds that a node issues in its lifetime $r = \frac{\lambda_s}{\lambda_f}$. Low r means less stabilisation and therefore higher churn. The effect of churn to Chord (studied in [71]) is mainly that the entire successor lists becomes outdated quickly for large churn rates resulting in degraded performance and inconsistent lookups. Under churn or in the presence of NATs the chain of successor pointers may not form a perfect ring. In this paper we will use the term *core ring* to denote the periodic chain of successor pointers inherent in the Chord protocol. Due to NATs, some nodes are not part of the core ring and are said to sit on a *branch.* Figure 8.1a depicts such configuration.

8.4 Lookup Consistency

One of the most important properties of any lookup mechanism is consistency. It is however a well known fact [46, 106] that lookup consistency in Chord is compromised when the ring is not perfectly connected.

If two nodes queries the system for the same key they should receive identical values but due to node churn a node's pointers might become outdated or wrong before the node had time to correct them in a stabilisation round. In any case where a peer wrongly assigns its successor pointer to a node not succeeding it, a potential inconsistency is created.

The configuration in Figure 8.1a can lead to a potential lookup inconsistency in the range $]q,\ r]$. Upon lookup request for a key in that range, peer q will think r is responsible for the key whereas peer p think peer s is.

These lookup inconsistencies due to incorrect successor pointers can be both due to NATs (a node cannot communicate with its successor) or due to churn (a node is not aware of its successor since it did not maintain the pointer).

A solution proposed by Chord's authors in [46] is to forward the lookup request to a candidate responsible node. The candidate will verify its *local responsibility* by checking its predecessor pointer. If its predecessor its a better candidate, the lookup is sent backwards until reaching the node truly responsible for the key.

In order to quantify how reliable the result of a lookup really is it is important to estimate the amount of inconsistency in the system as function of churn and other system parameters. Measuring lookup inconsistency has been considered in a related but not equivalent way in [106].

We define the *responsibility range,* Q as the range of keys any node can be hold responsible for. By the nature of the Chord lookup protocol a node is responsible for a key if its preceding peer says so. From a global perspective it is then possible to ask each node who it thinks succeeds it on the ring and sum up the distance between them. However, as shown in Figure 8.2, the mere fact that two nodes think they have the same successor does not lead inconsistent lookups.

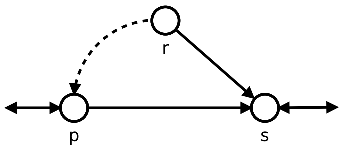


Figure 8.2: A lookup for a key owned by s will not give inconsistent results due to the fact that r is not a successor of another node.

In order to have an inconsistent lookup two nodes need to think they are the immediate predecessor of the key and have different successor pointers. In order to quantify the amount of inconsistency in the system we then need to find, for each node the largest separating distance between it and any node that has it as its successor. This value is the maximum range the node can be held responsible for. The sum of each such range divided by the number of keys in the system will indicate the deviation from the ideal case where $Q = 1$. The procedure is outlined in listing 8.1.

Listing 8.1: Inconsistency calculation

```
for n in peers do
    // First alive and reachable succeeding node
    m = n.getSucc()
    d = dist(n,m)
    // Store the largest range
    m.range = max(d,m.range)
end for

globalRange = sum(m.range)
```

The responsibility range is measured for various fractions of NATs as function of churn and the results are plotted in Figure 8.3. We see that even for low churn rates the responsibility range after introducing the NATs are greater than 1. This indicates that on average more than one node think they are responsible for each key which causes a lot of inconsistent lookups. Important to note is also that

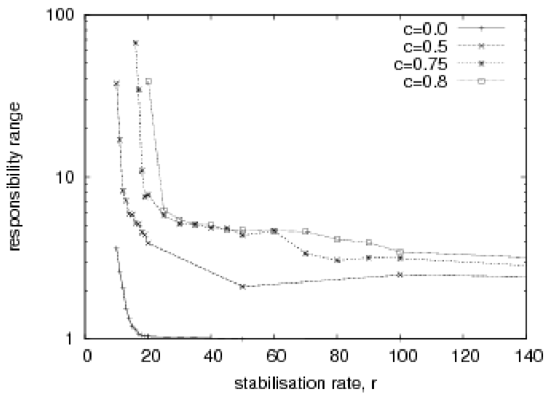


Figure 8.3: Responsibility range as function of stabilisation rate for various values of c. Even for low churn, NATs introduce a large amount of lookup inconsistencies as indicated by the range being greater than 1

without NATs (c=0) the churn induced inconsistency for low r is much higher than 1 .

Predecessor list (PL) routing

The result indicates that merely assigning the responsibility to your successor does not provide robust lookups in the presence of NATs or for high churn. Using *local responsibility* instead of the successor pointer to answer lookup requests, and routing through the predecessor will ensure that only the correct owner answers the query. Because more peers become reachable, there are less lookup inconsistencies, and more peers can use their local responsibility to share the load of the address space. Again, taking the configuration in figure 8.1a as example, we can see that peer q is unreachable in traditional Chord, and therefore, the range $]p,\ q]$ is unavailable. By using the predecessor pointer for routing, a lookup for key k can be successfully handled following path $p \to s \to r \to q$. However, by only keeping one predecessor pointer, lookups will still be inconstant on configurations such as the one depicted in Figure 8.1b.

To be able to route to any node in a branch, a peer needs to maintain a *predecessor list*. This is not the equivalent backward of the successor list, which is used for resilience. The predecessor list is used for routing, and it contains all nodes pointing to a given peer as its successor. If peers would use the predecessor list in the depicted examples, the predecessor list of s in Figure 8.1a would be $\{p,\ r\}$. The list in Figure 8.1b would be $\{p,\ q,\ r\}$. In a perfect ring, the predecessor list of each node contains only the preceding node. This means that the size of the routing table is not affected when the quality of the connectivity is very good. In the presence of NATs or churn, however, it is necessary to keep a predecessor list in order to enable routing on node branches . From a local perspective, a peer knows that it is the root of a branch if the size of its predecessor list is greater that one. Such predecessor list (PL) routing is used for instance in the relaxed-ring system [88] to handle non-transitive underlay networks.

In order to evaluate the performance of PL routing we define another responsibility measure, the *branch range, Q_b*. Since the PL routing protocol lets the root of a branch decide who is responsible (or whom to relay the lookup to) it is important that the root correctly knows the accumulated responsibility range of all nodes in its branch. The only way a PL lookup can be inconsistent is if two distinct root nodes on the same distance from the core ring think they should relay the lookup to one of its predecessors. The problem is depicted in Figure 8.4.

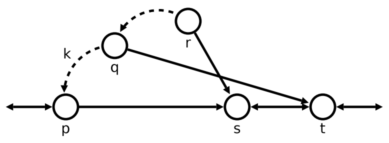


Figure 8.4: Peer q misses its successor s. Peer s routes *lookup(k)* to r because it does not know q.

The branch range is calculated in a similar way as the previous predecessor range. Each node, on the core ring, queries all its predecessors except for the last one (the one furthest away) for their range. The predecessors iteratively queries their predecessors and so forth until the end of the branch is reached. Each node then returns, among all the answers from its predecessors, the end point furthest away from the root. The root, in turn, compares the returned value with the key of its last predecessor. If the returned key lies behind the last predecessor the range can give rise to an inconsistent lookup. The procedure is outlined in listing 8.2.

Listing 8.2: Branch inconsistency calculation

```
// Peers in the core ring are first
// marked by cycle detection.
for n in corePeers do
    n.range = 0;
    for m in n.preds
      // Don't add pred from core ring
      if(m!=n.preds.last)
        n.range+=m.getBranchEnd(m.key)
    end for
    n.range+=n.preds.last
end for

function getBranchEnd(p)
   for m in n.preds
       p=min(p,m.getBranchEnd(m.key))
   end for
   return p
end function
```

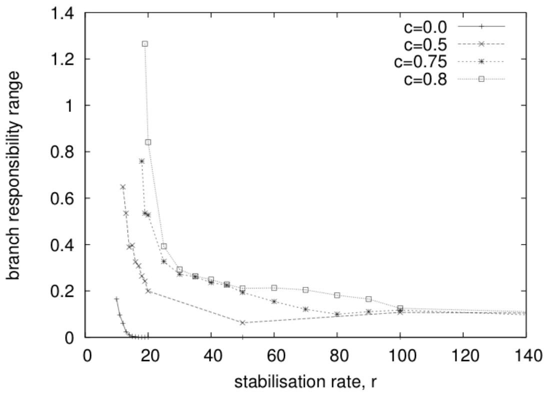


Figure 8.5: Branch responsibility range as function of stabilisation rate. As the system breaks down due to high churn the trees starts to overlap resulting in additional lookup inconsistencies.

`globalBranchRange = sum(m.range)`

The branch responsibility range, Q_b, includes the former range, Q, found without PL routing and adds any extra inconsistency caused by overlapping branches. Figure 8.5 shows the additional responsibility range induced overlapping branches. We see that the responsibility ranges resulting from overlapping trees are orders of magnitude smaller the ranges due to problems with successor based routing. Even in the worst case the range does not exceed 1.5 which means that at least 50% of the lookups will be reliable in worst case. Interesting also to note that the churn based inconsistency ($c = 0$) does not exceed 0.2 in any case which means the lookups are reliable to at least 80%.

Skewed key range

In Chord, it is mandatory for the joining peer to be able to communicate with its successor in order to perform maintenance.

When joining the system a node is provided a successor on the ring by a bootstrap service. If joining node is unable to communicate with the assigned successor due to a NAT all that remains to do is for the node to *re-join* the system.

As more and more NATed peers join the network the fraction of potential successors decrease creating a skewed key distribution range available for NAT nodes trying to join the ring. This leaves joining peers behind a NAT to join only closer and closer to the root.

As the fraction of NATed peers increase in the system additional join attempts are need in order to find a key succeeded by an open peer. The number of re-join

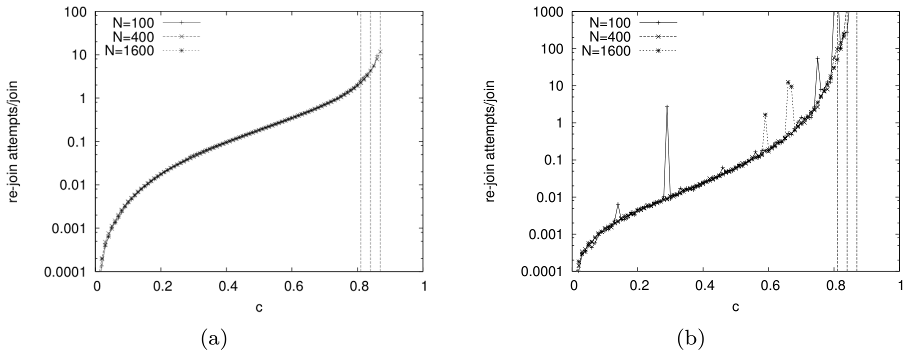


Figure 8.6: Distribution of number of re-join attempts as function of *c*. 8.6a Average number of re-join attempts before a node is able to join the ring for different values of c. Vertical lines indicate the point where a some node in the system needs to re-try more than 10.000*N times. 8.6b Variance of the number of re-join attempts.

attempts as function of *c* is shown in Figure 8.6a and the variance in Figure 8.6b. Note that both the average and the variance for the number of join attempts start to grow super exponential at some critical *c* value ≈ 0.8 which indicates a behavioural transition between functional and congested system state.

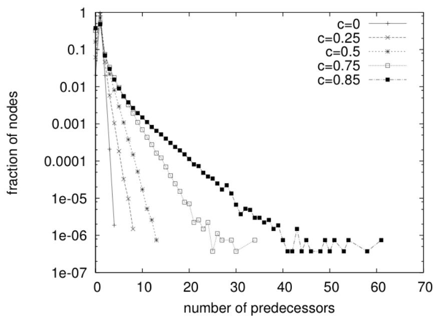
8.5 Resilience

Load balancing

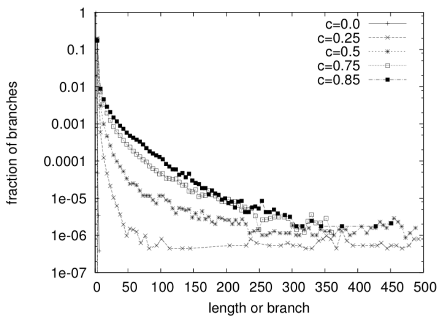
Even for the *c* range where nodes can join in reasonable amount of time the open nodes on the network will be successors for an increasing number of NATed peers. Being the successor of a node implies relaying its lookups and sending it successor- and recovery lists while stabilizing. Increasing the number of predecessors therefore increase the workload of open peers.

In the case of predecessor list (PL) relays, open nodes as branch roots will be responsible to relay lookups not only to itself and its predecessors but to all nodes in its branch. Figure 8.7a shows the distribution of the amount of predecessors per node and Figure 8.7b shows the size distribution of existing branches.

For large *c* values we note that some nodes have almost 1% of the participating nodes a predecessors and are the root in branches of size $0.2 * N$. When such a overloaded node fails a large re-arrangement is necessary at high cost.



(a)



(b)

Figure 8.7: The load on the open peers increases with c as they receive more predecessors and acts as relay nodes for the branches. The system contains 2^{12} nodes. 8.7a Fraction of nodes with a given number of predecessors. For large values of c the number for the open peers become orders of magnitude higher than on the NATed peers. 8.7b Size of branches in the system for various values of c. For low values trees are rare and they have small size. As the fraction of NATs grow the length of branches grow too.

Sparse successor lists

The size of the successor list, typically $log(N)$, is the resilience factor of the network. The ring is broken if for one node, all peers in its successor list are dead. In the presence of NATed peers the resilient factor is reduced to $log(N) - n$ for nodes behind a NAT, where n is the average number of NATed peers in the successor list. This is because NATed peers cannot use other NATed peers for failure recovery. The decrease in resilience by the fraction of NATed nodes is possible to cope with for low fractions c. Since there still is a high probability to find another alive peer which does not sit behind a NAT the ring holds together. In fact the NATs can be considered as additional churn and the effective churn rate becomes $r_{eff} = r(1-c)$

For larger values of c, however, the situation becomes intractable. The effective churn rate quickly becomes very high and breaks the ring.

Recovery list

As we previously mentioned, the resilient factor of the network decreases to $log(N)-n$. To remedy this a first idea to improve resilience is to filter out NATed peers from the successor list. However, the successor list is propagated backwards, and therefore, the predecessor might need some of the peers filtered out by its successor in order to maintain consistent key ranges.

We propose to use a second list looking ahead in the ring, denoted the *recovery list*. The idea is that the successor list is used for propagation of accurate information about peer order and the recovery list is used for failure recovery, and it only contains peers that all nodes can communicate with, that is, open peers. The recovery list is initially constructed by filtering out NATed peers from the successor list. If the size after filtering is less than $log(N)$, the peer requests the successor list of the last peer on the list in order to keep on constructing the recovery list. Ideally, both lists would be of size $log(N)$. Both lists are propagated backwards as the preceding nodes perform maintenance. Figure 8.8 shows the construction of the recovery list at a NATed peer.

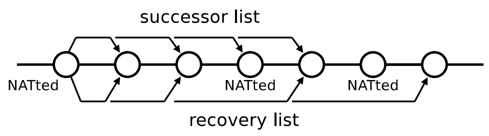
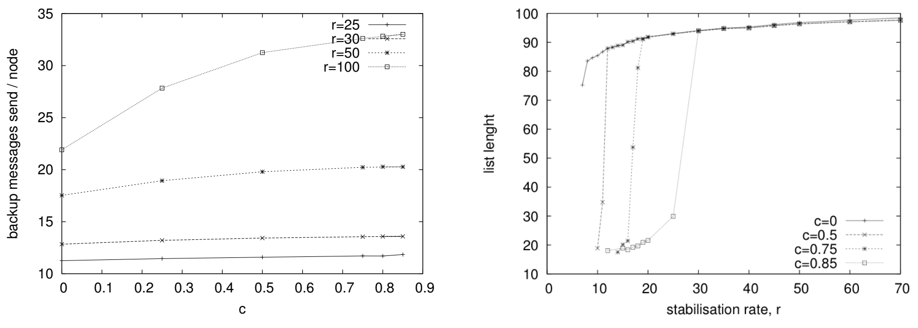


Figure 8.8: Two lists looking ahead: the successor and the recovery list.

Because both lists are propagated backwards, we have observed that even for open peers is best to filter out NATed peers from their recovery lists even when they can establish connection to them. The reason is that if an open peer keeps references on its recovery list, those values will be propagated backward to a NATed peer who will not be able to use them for recovery, reducing its resilient factor, incrementing its cost of rebuilding a valid recovery list, and therefore, decreasing performance of the whole ring. If no NATed peers are used in any recovery list, the ring is able to survive a much higher degree of churn in comparison to rings only using the successor list for failure recovery.

The recovery lists are populated in a reactive manner until it finds a complete set of open peers or can only communicate with peers with overlapping set of recovery nodes. The number of messages sent during stabilisation of the recovery list is shown in Figure 8.9a. For large stabilisation rates and high c values more messages are generated as the nodes need to make more queries in order to fill its recovery list.

Figure 8.9 shows how the average size of the recovery list varies with both churn r and fraction of NATs c. In the absence of NATs ($c = 0$) the recovery list maintains about the same size over the whole r-range. In the presence of NATs on the other hand, the size of the list abruptly decreases to only a fraction of its maximum. The reason for this effect is that for high enough churn large branches tends to grow. Large part of the ring are transferred to branches and while new nodes join they grow while the core ring shrinks. The small size of the recovery list reflects the fact that there are only a few open peers left on the core ring for high c and churn. Since the updates are done backwards and most open peers are situated in a branches nodes outside of the branch will not receive any information about their existence. The system can still function in this large branch state but becomes very sensitive to failures of open peers on the core ring.



(a) (b)

Figure 8.9: 8.9a Average number of backup messages sent on each stabilisation round by a node for different values of c as function of stabilisation rate. 8.9b Size of the recovery list for various c values as function of stabilisation rate. Before break-up there is a rapid decrease in number of available pointers.

8.6 Working range limits

Summarizing, we can categories the system behavior in the following ranges of the parameter c:

$0 \leq c < 0.05$. In this range the influence of NATs is minor and can be seen as an effective increased churn rate. If a NATed node find itself behind another NATed node it can simply re-join the system without to much relative overhead.

$0.05 \leq c < 0.5$. In the intermediate range the open peers are still in majority. Letting NATed nodes re-join when they obtain a NATed successor will however cause a lot of overhead and high churn. This implies that successor stabilisation between NATed peers need to use an open relay node. Multiple predecessor pointers are needed in order to avoid lookup inconsistencies and peers will experience a slightly higher workload.

$0.5 \leq c < 0.8$. When the majority of the nodes are behind NATs the main problem becomes the inability to re-join for NATed peers. In effect, the open peers in the network will have relatively small responsibility ranges but will in turn relay the majority of all requests to NATed peers. The number of reachable nodes in the NATed nodes successor lists decrease rapidly with churn. A separate *recovery list* with only open peers is needed in addition to avoid system breakdown.

$0.8 \leq c < 1$. In the high range the only viable solution is to let only open peers participate and have the NATed nodes directly connected to the open peers as clients. The open peers can then split workload and manage resources among its clients but are solely responsible for the key range between it and the first preceding open peer.

To make the system function for even higher churn rates and NAT ratio, our conclusion is that one should only let open peers join the network and then attach the NATed peers evenly to them as clients. Since NATed peers do more harm than good, if there are to many of them we see no other option than to leave them out. Since the open peers get most (or all) of the work load, in any case it is better to spread it evenly.

8.7 Conclusions

In this work we have studied a model of Network Address Translators (NATs) and how they impact the performance of ring based DHT's, namely Chord. We examine the performance gains of using *predecessor based* routing and introducing a *recovery list* with open peers. We show that adding theses elements to Chord makes the system run and function in highly dynamic and NAT constrained networks. We quantify how the necessary adjustments needed to perform reliable lookups vary with the fraction of nodes behind non-traversable NATs.

We also note that having NATed nodes per se does not dramatically increase the probability of system failure due to break-up of the ring, as long as nodes behind non-traversable NATs can use some communication relay. The main reason why the ring eventually fails due to large churn is that the branches becomes larger than the length of the successor- and recovery list. Information about new peers in the ring cannot then reach nodes at the end of the branch who's pointers will quickly become deprecated. At the same time, as branches grow large, new nodes will have a high probability of joining a branch instead of the actual ring which will worsen the situation further in a feedback that eventually breaks the system.

Our conclusion is that it is indeed possible to adjust Chord, and related ring based DHT protocols, to function in the presence of NATs without the need for traversal techniques. It further shows that it is possible to construct robust overlay applications without the assumption of an open underlying network.

8.8 Acknowledgements

The authors would like to thank Jim Dowling for valuable comments and discussion. The project is supported by SICS Center for Networked Systems (CNS) and the European Community's Seventh Framework Programme (FP7/2007-2013) under Grant agreement n214898. The Mancoosi Project.

Chapter 9

On the Effects of Caching in Access Aggregation Networks

John Ardelius[1], Björn Grönvall[1],

Lars Westberg[2] and Åke Arvidsson[2]

[1] Swedish Institute of Computer Science (SICS), Sweden
`[john,bg]@sics.se`

[2] Ericsson Research, Sweden
`[Lars.Westberg,Ake.Arvidsson]@ericsson.com`

Abstract

All forecasts of Internet traffic point at a substantial growth over the next few years. From a network operator perspective, efficient in-network caching of data is and will be a key component in trying to cope with and profit from this increasing demand. One problem, however, is to evaluate the performance of different caching policies as the number of available data items as well as the distribution networks grows very large.

In this work, we develop an analytical model of an aggregation access network receiving a continuous flow of requests from external clients. We provide exact analytical solutions for cache hit rates, data availability and more. This enables us to provide guidelines and rules of thumb for operators and Information-Centric Network designers.

Finally, we apply our analytical results to a real VoD trace from a network operator and show that substantial bandwidth savings can be expected when using in-network caching in a realistic setting.

9.1 Introduction

Internet traffic forecasts point at a substantial growth over the next few years and a major part of this traffic is expected to be related to video [26, 56].

From an operators' perspective, a problem is to manage this new traffic at a reasonable cost. A possible solution to this problem is the introduction of in-network caching as has been proposed by the Information-Centric Networking (ICN) community. However, it shall be remembered that in-network caching has benefits not only to operators but also consumers and content providers may benefit from it.

While the amount of video related traffic is increasing, so is the number of distinct available items. Recent studies show, *e.g.*, that there are on the order hundred million different items available in BitTorrent networks [36]. Performance analyses of systems handling such large amounts of data are inherently difficult and mostly lacking.

The purpose of this paper is to introduce a tool or analytical model of in-network caching for truly large scale content catalogues. The tool is then used to gain insights, provide guidelines and, make predictions on the effects of in-network caching that are of importance to the designers of future ICN architectures.

Our aim is to provide a simple tool for assessing the load on caches and links at various levels in the network as well as estimating the availability of data. We remark that an in-network caching architecture must also account for the complex trade-offs between different kinds of hardware solutions and their associated costs. Our tool can provide valuable insights in this decision process.

The main contribution of this work is an analytical model for the performance of caches in truly large scale hierarchical access aggregation networks. The model is applicable to any cache eviction policy and we study two benchmark policies which yield estimates of:

- The hit rate at any level in the cache hierarchy as well as for the access network as a whole for networks containing of the order 10^8 distinct items.
- The specific content held at caches at various levels in the cache hierarchy.
- The overhead costs due to redundant caching.
- The performance gains from varying cache sizes.

Finally, based on traffic data from the network operator Orange, the tool is used in a realistic setting to show substantial bandwidths savings when in-network caching is applied to Video-on-Demand traffic (VoD).

9.2 Background and scope

Access aggregation networks and cellular back-haul networks typically form tree-like topologies. In fixed networks traffic ingress and egress tends to be concentrated to the leaves and the root of the tree. In contrast to this, cellular base stations are often co-sited with nodes in the tree and traffic may enter or leave at any node in these networks.

It is hard to give a specific number of child nodes per parent node in the tree as this number depends on the underlying transmission technology and the dimensioning margins for the specific traffic mix, but a typical range is 3–10 and we note that our model is flexible in this respect.

Because of space limitations, the discussion will be limited to access aggregation networks in the fixed domain leaving cellular networks for future work.

In general, we cannot predict topologies or usage patterns of future networks. Nevertheless, we will make the assumption that in-network caching can be deployed at any node in our tree shaped networks.

9.3 Related work

The efficiency of caches has previously been subject to benchmark studies in the context of disk accesses [62].

Analytical studies of caches also exist, in [35] the hit rate performance of LRU is studied using an approximate model for a single cache. Our approach is somewhat different in that we are studying the exact solution and in the context of a hierarchy of caches. Other work such as [100] analyze worst (and best) case behavior of caches whereas we are interested in the average case which will dominate for truly large systems.

The performance of LRU caches has also been studied by Che *et al.* in [24] and in the context of hierarchical caching in [47], where a very accurate hit rate approximation per content is given. However, solving the approximation equation for millions of items is neither straight forward nor feasible in practice.

A performance analysis of a two-layer hierarchy using the Che approximation is given in [48] for moderately sized (10^4 items) systems.

Caches operating under continuous data flux have been studied in the context of automatic control in [120], again limited to single caches.

The performance of web caches has been well studied experimentally and using simulations. Few analytical studies treating explicit eviction policies exist, however. A nice presentation of some general web caching techniques can be found in [117].

9.4 The network model

We model an access network as a graph of interconnected nodes (routers). Each node may have any number of clients (hosts) attached which request and receive content. Clients can only be connected to one node at the time. Nodes can either serve requests immediately or relay them to other nodes in the network.

To simplify the analysis we approximate the connection topology of the nodes by a d-regular tree. Leaf nodes at the bottom resemble access routers far out in the

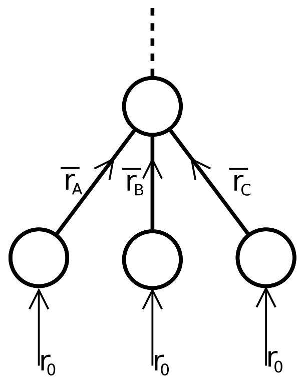


Figure 9.1: Each leaf node receives a vector of external requests r_0 and relays the requests not served by its cache $\bar{r}_k$ to its parent node.

network whereas the root is a member of the backbone or core network providing connectivity to the Internet.

Due to the tree structure of the network, a node is connected to one node higher up in the tree and d nodes below except for the leaf nodes at the very bottom which only are connected upwards. The *level* of a node indicates its distance plus one in hops from the leaf level hence the root will have the highest level equal to the depth of the tree and the leafs will be on level one. Each node also has a storage capacity or *cache size* C which determines the number of cache items it can store locally.

Further we assume a fixed, large catalogue of N items which can be requested by any node in the network. Content requests can originate either from clients connected to the node, or from other nodes in the network which are unable to service the request from their local caches. In more detail, a node serves all requests for which the content resides in its cache, and relays all other requests to its parent node. This is depicted in Fig. 9.1.

Continuous content flow

Rather than examining the behavior of the system at each individual request, we are interested in the overall *average* performance of the system as function of a few key parameters. Given a probability distribution for the content request rates, an eviction policy for the local caches and a particular network topology, we would like to determine the bandwidth usage and content availability for the entire network.

In order to do this, we study the system in the limit where the number of requests per time unit is so large that we may approximate the arrival process of discrete requests by a continuous *flow* which is specified as a rate of requests per unit time. This approach is similar to fluid analysis of queuing systems [68]. Our analysis then operates on distributions of these flows rates rather than on discrete request events. We thus assume that there exists a period of time where the distribution of content requests is approximately static. We further assume that the exchange rate of availble items is such that it is approximately static as well.

In the continuous limit we define a *request vector* $\mathbf{r}$ as a vector of average request rates for each available item,

$$\mathbf{r} \equiv \{\mu_0, \mu_1, \ldots, \mu_N\}.$$

We further assume that two requests for a particular piece of content are independent which motivates us to model the continuous process of requests for item i as a Poisson process with rate λ_i. This is indeed a simplification, but it will provide us with a lower performance bound since correlation among requests can be exploited by a clever caching strategy.

The requests received by a particular node k can then be split up in two parts; one which contains requests from local clients, $\mathbf{r}_0$, and another one which contains requests relayed from other nodes in the network, $\bar{\mathbf{r}}_k$,

$$\mathbf{r}_k = \mathbf{r}_0 + \bar{\mathbf{r}}_k. \tag{9.1}$$

As suggested in various studies, *e.g.* [22], the aggregate request intensities from external clients can be fairly well approximated by a power law distribution. Here we assume that intensities follow a Zipf distribution with parameter α which determines the shape of the power law. Typical values for the popularity of Internet content lies in the range $\alpha = 0.6 - 0.9$ [22, 36].

$$\lambda_i(i, \alpha, N) \propto \frac{1/i^\alpha}{\sum_{n=1}^{N}(1/n^\alpha)} \equiv p(i) \tag{9.2}$$

The item indexes are ordered by decreasing popularity where item 1 is the most popular.

The first term in equation (9.1) represents requests issued by clients *locally* at the node itself and is obtained as

$$\mathbf{r_0} \equiv \{\lambda_0, \lambda_1, \ldots, \lambda_N\}$$

where the rates are given by the Zipf distribution (9.2).

The second part of equation (9.1) represents the flows relayed from other nodes n,

$$\bar{\mathbf{r}}_k \equiv \sum_{n \in M_k} \bar{\mathbf{r}}_{n \to k}. \tag{9.3}$$

where M_k is the set of nodes which relay requests to node k.

Further, we simplify the analysis by assuming that the network is well balanced (which is only approximately true in practice). The assumption lets us treat each node on the same level interchangeably,

$$\mathbf{r}_i = \mathbf{r}_j; \forall \{i, j : \text{level}(i) = \text{level}(j)\}. \tag{9.4}$$

This means, due to symmetry of the tree, that the *average* request pattern seen by a node at level x is the same for all nodes at this level.

The probability that a node k holds an item in its cache is given by the vector

$$\mathbf{c}_k \equiv \{c_k^0, c_k^1, \ldots, c_k^N\}; c_k^i \in [0, 1].$$

9.5 Eviction policies

There are two basic cache eviction algorithms known as LFU (Least Frequently Used) and LRU (Least Recently Used). Many modern eviction algorithms are variations of these two and use combinations of *frequency* and *recency* in their implementation.

The LFU eviction policy is to remove the item with the smallest number of accesses. Unfortunately this means that the algorithm can not be implemented in constant time hence it is therefore mostly of theoretical importance when applied to large caches.

The other, much simpler policy is LRU. It evicts the item that has not been accessed in the longest time. LRU can easily be implemented in constant time using a double linked list.

Our main concern in this paper is to provide benchmark points for our analytic model. For this reason we provide analytic expressions for the performance of both LFU and LRU under continuous content flow.

Thus, we study the system in steady-state and do not consider temporal variations. This will lead to conservative performance estimates, since variations in principle are exploitable by clever eviction policies.

Least Frequently Used (LFU)

In our model, when the cache eviction policy is based on frequency information, the node will look at its incoming request vector and compare the frequency of request for the items held in the local cache with requests for other items. The best option for the node, regarding its request vector $\mathbf{r}$ is then to cache the items with maximal average request rate. Let $F_C(\mathbf{r})$ be a function that returns the C largest values from $\mathbf{r}$. The cache probability vector for node k, $\mathbf{c}_k$ will then be given by.

$$\mathbf{c}_k^{\text{LFU}} = \mathbf{1}_{i \in F_C(\mathbf{r}_k)}$$

Where $\mathbf{1}$ is the indicator function. The intuitive interpretation of this LFU model is that if average request rates are static and known, the best policy in steady state is to cache items with high average request rates. This is not however an optimal strategy since temporal variation in requests can motivate eviction of items which are popular in the long run. This LFU model will thus provide us with an average case estimate rather than an upper or lower performance bound.

Least Recently Used (LRU)

In order to calculate the probabilities for a node k to have an item a in its cache while using the LRU policy we can do this by calculating its complement, namely the probability that the item will be evicted. The eviction probability is determined by the probability the cache receives a sequence of requests **not** containing a while at least $C-1$ other items are requested. We denote this probability by R(a).

Consider a sequence of requests of length s. A *configuration* of s is a vector $K = \{k_1, k_2, \cdot, k_N\}$ with the specific number of requests, k_j, issued for item j in this sequence. $\sum_j k_j = s$. If we denote the set of items different from a by $\bar{a}$, the probability for a specific configuration K of the s requests among the items $\bar{a}$ is given by the multinomial distribution

$$P(\bar{a}, K, s) = \frac{s!}{\prod_{i\in\bar{a}} k_j!} \prod_{i\in\bar{a}} (p(i))^{k_j} \tag{9.5}$$

Summing over all s we get

$$\sum_{i=C}^{\infty} P(\bar{a}, K, i) = \frac{(\sum_{j\in\bar{a}} p(j))^C}{1-(\sum_{j\in\bar{a}} p(j))} \tag{9.6}$$

This is the probability that item a will not appear in the sequence but we also need to make sure that at least $C-1$ other items are requested. This will then cause item a to be evicted.

Let $Q_j(a)$ denote the set of all unique sequences of length s where only j unique items are requested. The probability for such a subset is again given by equation (9.6). We then need to subtract all such sets where less than $C-1$ unique items are requested since these sets will for sure not cause a to be evicted. However, in each $Q_j(a)$ we will over count a number of subsets of shorter length $j-1, j-2, \cdots$. Therefore, the correct probability is given by the inclusion-exclusion equation below. Taken together the eviction probability for an LRU cache is given by:

$$R(a) = p(a)(\frac{(1-p(a))^C}{p(a)} - Q(a)) = (1-p(a))^C - p(a)Q(a) \tag{9.7}$$

where

$$Q(a) = \sum_{i=1}^{C-1} Q_i(a) + \sum_{j=1}^{i-1} (-1)^{j+1} \binom{C-(N-1)}{N-1-j} Q_j(a) \tag{9.8}$$

The probability that a given item resides in cache is then given by:

$$\mathbf{c}_k^{\mathrm{LRU}} = \frac{1-R(k)}{\sum_j 1-R(j)} \tag{9.9}$$

Equation (9.9) can, for moderate number of items and cache size, easily be verified by simulation.

For very large numbers of available contents the exact solution is intractable due to the fact that we need to calculate each term in eq (9.8). However, since each $Q(a)$ only contains at most $N-1$ terms, each less than 1 we will when $C >> 1$ be able to approximate the cache probability by

$$c(k) \simeq 1-(1-p(a))^C \tag{9.10}$$

9.6 Bottom-up request routing

In the following, content will not be allowed to flow up the tree. Due to the symmetry property of the request vectors (Eq. (9.4)), we denote the requests from a node at level j, which is same for all nodes at this level, by $\mathbf{r}^j$. Further, due to the regular degree of the tree, the residual flow equation (9.3) simplifies to

$$\bar{\mathbf{r}}^j = d\bar{\mathbf{r}}^{j-1} \tag{9.11}$$

The nodes at level 1 (the leafs) will in the bottom-up case only see requests from external client $\mathbf{r}^1 \equiv \mathbf{r}_0$ since no relay traffic reaches the leafs.

The residual flow out of level j is the flow not absorbed by the cache at that level and is given by

$$\bar{\mathbf{r}}^j = \mathbf{r}^j \cdot (1-\mathbf{c}^j) \tag{9.12}$$

Using equation (9.4) again, the request vector for nodes at higher levels becomes

$$\mathbf{r}^j = \mathbf{r}_0 + d\bar{\mathbf{r}}^{j-1} = \mathbf{r}_0 + d(\mathbf{r}^{j-1} \cdot (1-\mathbf{c}^{j-1})) \tag{9.13}$$

Solving this recursion relation using equation (9.11) and (9.12) gives

$$\bar{\mathbf{r}}^j = \mathbf{r}_0 + \sum_{i=1}^{j} d^{i-1} \prod_{n=j+1-i}^{j} (1-\mathbf{c}^n) \tag{9.14}$$

and the residual flows

$$\bar{\mathbf{r}}^j = \mathbf{r}_0 d^{i-1} \prod_{n=1}^{j} (1-\mathbf{c}^n). \tag{9.15}$$

9.7 Results and observations

Next we solve the equations for a tree of degree $d = 10$ and a three levels for various catalog sizes N and cache sizes C. Requests will be Zipf distributed using $\alpha = 0.7$.

In Figure 9.2 we plot the aggregate hit rate for the whole cache hierarchy for three different catalogue sizes $N = 10^3, 10^4$ and 10^5. In each setup we vary the cache size from 0.1 to 10 percent of the catalogue size. We note that the hit rate of LFU is almost twice that of LRU and that the benefit of increasing cache size

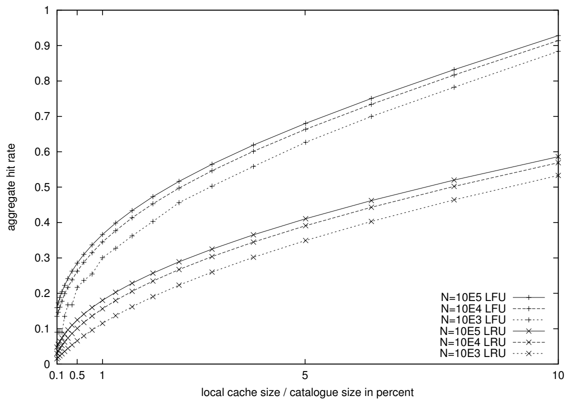


Figure 9.2: Aggregate hit rate as function of cache size for LFU and LRU.

grows faster. In both cases we note a diminishing return of cache size to hit rate as the catalog grows large. Cache size has to grow super linear in order to achieve a linear gain in hit rate. This is also consistent with observations made by others e.g [22].

Figure 9.3 shows the hit rate of individual levels for a very deep tree network of 10 levels and a cache size of 0.1% ($C/N = 0.001$). We note that the first level (leaf) cache is the most important one. Caches at higher levels can still contribute to the aggregate cache effect, but only to a much smaller extent (this is not additive). Thus, when designing future in-network caching architectures, one should consider using either larger caches at higher levels or creating groups of collaborating creates to form larger virtual caches (paying the internal communication costs but increasing content availability). This holds for the whole parameter range but the relative performance of level 1 decreases with catalog size. This is intuitive since, for a very large N, each level in the cache will store very popular items and will be of similar importance to performance.

Next we examine what items will likely be cached by the two eviction policies. A rank plot of popular items and the probability of finding them cached is given in Figure 9.4. Here we see the difference between LRU and LFU clearly. LFU caches *only* the most popular items (head of the distribution) whereas LRU will cache all items with a non zero probability. An interesting effect of this is that if caches did collaborate, LFU would still have to request many items from outside of the tree whereas LRU could almost surely find the item somewhere locally. Thus, collaborative caching can also be used to improve availability in the case of network problems or failures of external links.

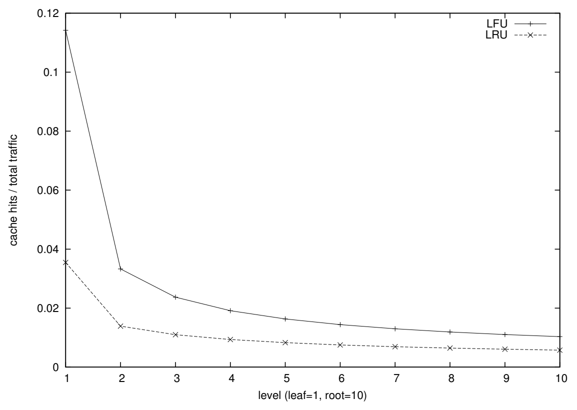


Figure 9.3: Hit rate for various levels in the cache hierarchy. The leaf level is denoted 1.

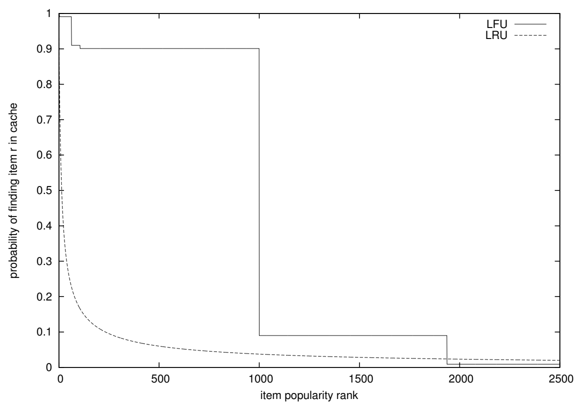


Figure 9.4: Comparison between which items are cached by LRU and LFU by rank.

A realistic example

Next we will apply the analytic model on a fictitious access aggregation network. The network will consist of three levels of routers each with a fan-out (degree) of 10. The topmost (root) router is directly connected to the operators' metropolitan network and the remaining routers form a tree below the root. A possible interpretation of this sample network is one hundred Digital Subscriber Line Access Multiplexer (DSLAM) pools that in turn are connected to ten Broadband Remote Access Servers (BRAS) that are conneced to the root router.

Each router is equipped with 360 Gbytes of flash memory to be used for in-network caching. We pick 360 Gbytes because there is a standard PCI-Express card that currently costs 700 EUR with this amount of memory. The card provides 540 Mbytes/s of sustained read capacity. For simplicity all routers are equipped with the same amount of cache memory.

Duration	8 days		
Requests	29,707	5,199 clients	5.7 reqs/client
Catalog	3,518 items	2,473 GB	703 MB/item

The average content size is 703 Mbytes which with a 360 Gbyte cache corresponds to a cache capacity of 512 items. In this example we want to calculate lower bounds on cache hit rates and will consequently use the LRU eviction policy, which in our model does not benefit from correlations in the requests stream.

With these parameters as input to the analytic model we expect a hit rate of 45% at the routers closest to the terminals. At the next level hit rate will be 26% and at the root 22%. Taken together this represents an aggregate cache hit rate of 68%, i.e only 32% of the VoD traffic is expected to be visible outside the tree.

The results (hit rates) of the calculations are summarized for three different xache sizes below.

Level	256GB	360GB	512GB
Root router	0.184	0.218	0.260
Middle (BRAS)	0.223	0.264	0.315
Leaf (DSLAM)	0.376	0.448	0.523
Total	0.604	0.683	0.758

An important question to consider is if flash memory will provide enough capacity to satisfy all read traffic.

To this end, we first note that from the trace data we can deduce an average VoD rate of 30 Mbyte/s of which 45% or 13.5 Mbyte/s should be read from our cache. In our experience, peak VoD load can be as much as 5 times the average load. In this case, there is plenty of spare capacity to satisfy the demand.

Similarly, at the root level we must read data from the cache at an average rate of 265 Mbyte/s. Although this value may be possible, the same does not apply to peak loads as much as 5 times the average load, in which case we need to use flash memory with higher read performance. Such memory exists and considering that this router should be a high end one, the additional cost can likely be motivated.

In June 2011 Cisco reported [26] *Internet video is now 40 percent of consumer Internet traffic, and will reach 62 percent by the end of 2015, not including the amount of video exchanged through P2P file sharing. The sum of all forms of video (TV, video on demand [VoD], Internet, and P2P) will continue to be approximately 90 percent of global consumer traffic by 2015.*

If we assume Internet video accesses to be similar in character to our VoD data, it should be within reach to save some 27% of all consumer Internet traffic and this should extend to 42% savings by the end of 2015. Also, remember that our model is conservative and does not benefit from request correlations. Bandwidth savings should thus be expected to be larger than this.

9.8 Conclusion and future work

In this paper we have developed an analytical model of an access aggregation network with in-network caches and we have given exact analytical expressions for the performance of LRU and LFU caches. Using these results we were able to derive probabilities that given items reside in particular caches, determine cache efficiencies in terms of hit rates and network loads. The model enabled us to study systems scaling to millions of data items and thousands of nodes, something that is often impossible using simulations.

In the context of Zipf distributed accesses we have observed that LRU is better at caching items across the catalog making it a better candidate for collaborative caching and improvements in availability. On the other hand LFU can achieve better cache hit rates.

For practical purposes, one needs to strike a balance between frequency and recency in the eviction policies to meet expectations of availability and cache hit rate. The analytical model can aid in striking this balance.

Finally, in a realistic example we showed that it should be possible to save 68% of VoD traffic using inexpensive caches. As of today this corresponds to approximately 27% of all consumer Internet traffic [26].

Future directions of this work will be to analyze other eviction policies using the same basic, model and to extend it to study the impact of intra network cache collaboration. We intend to extend the analysis to more realistic request patterns containing temporal correlations among requests as well as time varying popularity distributions.

Chapter 10

Towards A Theory of Tree-Based Aggregation in Dynamic Networks

Supriya Krishnamurthy[1], John Ardelius[1], Erik Aurell[2]
Mads Dam[2], Rolf Stadler[2], Fetahi Zebenigus Wuhib[2]

[1] SICS Center for Networked Systems (CNS), Sweden
`[supriya,john]@sics.se`

[2] ACCESS Linnaeus Center, KTH - Royal Institute of Technology, Stockholm
`[eaurell,mfd,rolf.stadler,fetahi]@kth.se`

Abstract

Tree-based protocols are used for a large variety of purposes in distributed systems. In the static case, tree-based protocols are generally well-behaved and easy to analyze. In the dynamic case, however, the addition and removal of nodes can have complex global effects on the tree overlays. We present a Markovian analysis of the accuracy of tree-based counting under churn. The model is constructed in two stages. First, we derive a continuous-time Markov model which reflects the distribution of nodes over the levels in the aggregation tree and derive an exact equation for the expected value of the occupation of each level. We solve this equation with the help of a key Lemma (which we prove under certain conditions) and which we believe is of independent interest, namely that for each tree level the in-flux and out-flux of nodes, due to network dynamicity, balances.This result now also allows other key protocol characteristics to be estimated, including the expected number of nodes at a given (perceived) distance to the root and, for each such node, the expected (perceived) size of the sub-network rooted at that node. We validate the model by simulation, using a range of network sizes, node degrees, and churn-to-protocol rates, with very good results. We also comment on possible connections with the theory of queueing networks.

10.1 Introduction

Trees are used for a huge variety of purposes in networking and distributed computing, including distributed data structures, routing, distributed data management, network monitoring, fault tolerance and fault management, content distribution, and information aggregation. In purely static settings the correctness and performance of tree-based algorithms is often quite simple. However, as networks grow larger, the assumption of static networks becomes increasingly untenable to justify, and nodes dynamically joining and leaving the network (in P2P terminology: churn) becomes a fact of life.

This is a challenge for algorithms design as well as analysis. Distributed data structures such as trees or DHT's (Distributed Hash Tables) typically need to maintain some form of invariant to remain functional. For instance, a distributed search tree needs to maintain invariant properties on keys occurring in various subtrees, a distributed BFS tree must maintain the invariant that paths from the root to any given node is of minimal length, and a DHT must maintain invariants related to its "finger" structure. As nodes join or leave, these invariants are temporarily broken, as the maintenance algorithms detect and recover from node deletions and additions. This ongoing process of invariant violation and repair due to churn causes the algorithms to produce erroneous output. The question we pose is: How erroneous?

As a case in point consider self-stabilizing algorithms [39]. Self-stabilization means roughly that the data structure is able to recover from arbitrary disturbances, provided churn is from some time onwards absent. This is a highly desirable property of a fault tolerant system. However, a large system may never be entirely free of churn, and in this situation, self-stabilization does not apply. In spite of this, for linear systems, conclusions may still be made, due to superposition. Distributed data structures are often, however, highly non-linear systems and the development of analytic methods for this type of system therefore presents a significant challenge.

We study the problem in the context of a simple tree-based aggregation protocol GAP [34], which is similar to the TinyOS TAG protocol [86]. GAP is a standard adaptation of the well-known Bellman-Ford algorithm [13] for aggregation, see also the self-stabilizing BFS maintenance algorithm of Dolev et al [39]. The Bellman-Ford algorithm solves the single-source shortest path problem by iteratively updating for each node a distance metric x, indicating the number of hops needed to reach a distinguished source node.

A classical application of Bellman-Ford is distance-vector routing. For aggregation we assume a fixed global root node, a management station. Each node maintains an estimate—its level—of the minimal number of hops needed to reach the root node. Each node then appoints a parent of lower level and uses the thus induced BFS tree overlay as a backbone for aggregation.

Self-stabilization of the GAP protocol is elementary. The root is always stable by definition. After 1 round all neighbours of the root are stable. After one more round, all grandchildren, and so on. Under churn, however, the matter is more complicated.

In this case the aggregation tree continually finds itself under reconfiguration: The removal of a node in the aggregation tree close to the root, can cause a large fraction of the entire aggregation tree to be orphaned at one go, creating large instantaneous errors. Some of the orphaned nodes may be able to maintain their tree level by reconnecting to alternative nodes at the parent level. Other orphaned nodes may need to reconnect at a higher level (levels further away from the root) in the tree. Some nodes may not find a path to the root at all, and so join a disconnected cluster of nodes with divergent levels (This is the well-known count-to-infinity problem in distance vector routing). Similarly, when nodes join the network, large parts of subtrees may be raised to levels closer to the root, and orphaned nodes may be recaptured.

This process of tree churn can cause very large transient errors. Transient behaviour is therefore not very informative as a performance measure. For this reason, in this paper we focus on average error. We model the protocol behavior using a continuous time Markov process. This process exactly captures the distribution of nodes over the levels in the aggregation tree but is not analytically tractable in general. We can however derive an exact rate equation for the rate of change of the expected value of the occupation of level x.We solve this equation in the steady state using a key Lemma (which we prove) which we believe is of independent interest, namely that for each tree level the in-flux and out-flux of nodes, due to network dynamicity is in equilibrium. The construction allows key protocol characteristics to be estimated, including the expected number of nodes at a given (perceived) distance to the root and, for each such node, the expected (perceived) size of the sub-network rooted at that node.

The model is evaluated through simulation, for network sizes ranging from 10^3 to 10^5, and for a range of node degrees. A key performance parameter is the rate r at which protocol state is updated per churn event where a churn event is a node joining or leaving the network.

In the case of $r = 1$ the tree overlay is highly unstable, and each node updates its state only once on average before it fails. For $r = 1,000$, tree-maintenance is performed at a much higher rate, and the overlay is consequently much more stable. The simulations allow us to make interesting conclusions regarding the performance of the GAP protocol under churn. Essentially we find that accuracy degrades significantly with the size of the network, with decreasing update rate r, and with decreasing node degrees. More importantly, however, we see that simulated behaviour is accurately predicted by the analytic model. This was our main objective, and it opens up for interesting new directions, such as using the analytic model to tune monitoring accuracy against update rate.

Related Work Churn has been most well-studied analytically in the context of queueing networks [67, 63, 119, 94, 112]. We comment in Section 10.7 on the connection of our model with a network of $M/M/\infty$ queues.

More recently there has also been a tremendous interest in churn and its implications on system performance, in the context of P2P systems. A large number of authors have used simulations [103, 81, 15, 50] and/or experimental data [55, 110] to study the effect of churn in these systems.A number of analytical studies have also been carried out in this context [83, 69, 50, 5, 118, 122].

In [71], Krishnamurthy et al used a model of churn similar to the one used here, to analytically estimate the performance of the Chord DHT under churn. The protocol studied here is far simpler and so we are able to model it at a much finer level of granularity than in [71].

The effects of churn on network connectivity have been studied by Leonard et al [79]. They relate node lifetime distributions to the likelihood that nodes get isolated, and estimate the probability of p2p system partitioning. Kong et al [70] study reachability properties of DHT systems and obtain analytic models of connectedness in terms of system size and node failure probability.

Many analytical studies of dynamic networked systems use so called fluid models [68]. These are generally used to analyse the macroscopic behaviour of a discrete system in the large scale (continoum) limit and have been successful in describing key properties of various systems such as BitTorrent-like networks [98], ad hoc mobile networks [76] and P2P streaming systems [75].

There are however surprisingly few prior works which analyse the performance of tree-based protocols under churn. In the context of multicasting, van Mieghem [91] reports on the stability of multicast shortest-path trees by estimating the expected number of link changes in the tree after a multicast group member leaves the group. The analysis is done for random graphs and k-ary trees. Lower bound results have been obtained for Demmer and Herlihy's Arrow protocol [37, 74], and the average case communication cost of Arrow is studied in [97], but only for the static case.

Structure of Paper Section 10.2 describes our network model and Section 10.3 an outline of the GAP aggregation protocol. In Section 10.4 we outline how we describe GAP by a continuous time Markov process, derive a simple and exact rate equation for the number $E[N_x]$ of nodes with a certain distance x to the root, and solve the rate equation by using a key Lemma which we prove. We then use some of these results to estimate the average partial aggregate a_x held by a node at level x in Section 10.5. Section 10.6 presents the simulation results, Section 10.7 a discussion on some interesting aspects and open questions related to our work and Section 10.8 a summary. Further details of the calculations are relegated to the appendix.

10.2 The Network Model

We model the network as a family of random graphs $G(t) = (V(t), E(t))$ indexed by time t. All nodes except the root (which is fixed), join the network by a Poisson process with intensity λ_j and fail independently with (per node) failure rate λ_f. When nodes join they do so with an *a priori* degree distribution $Q(k)$, which is

Poisson with average $\overline{\lambda}$ *i.e.* $Q(k) = e^{-\overline{\lambda}} \frac{\overline{\lambda}^k}{k!}$. The new joinee attaches to k different nodes (chosen with uniform probability) present in the network at the time of joining . The following observation is standard for birth-death processes [45] adapted to the present setting.

Proposition 10.2.1 *Assume a network following the above conventions, and let* $\lambda_j = N\lambda_f$.

1. *The a posteriori degree distribution is identical to* $Q(k)$.
2. *The expected network size (number of nodes) is* N.

10.3 The GAP Protocol

The protocol we analyze is based on the BFS algorithm of Dolev, Israeli, and Moran [40]. The graph has a distinguished root node v_0 which never fails, and which executes a program slightly different from the others.

A node $v \in V(t)$ has three associated registers lvl_v, agg_v and par_v holding the following values:

- $lvl_v \in \mathbb{N}^+ \bigcup\{\infty\}$ holds v:s *l*evel.
- $agg_v \in \mathbb{N}^+ \bigcup\{\infty\}$ holds v:s *p*artial aggregate. This is node v:s current belief of the aggregate contained in the subtree rooted in v.
- $par_v \in \bigcup\{v' \in V(t') \mid t' < t\}$ indicates the node (which may no longer exist) which v believes to be its parent in the tree.

A *c*onfiguration is a graph $G(t)$ along with assignments to the three registers for each node in $V(t)$. We write $lvl_v(t)$ when we want to refer to the value of lvl_v at time t and similarly for the other registers.

Each node randomly and independently executes *updates*, which are Poisson distributed events with rates λ_g. Let $n_v(t)$ be the set of neighbours of v at time t. When one node v executes an update at time t its registers are updated as follows:

- $lvl_v := \min\{lvl_{v'} \mid v' \in n_v(t)\} + 1$, where $\min(\varnothing) = \infty$ and $\infty + 1 = \infty$.
- $agg_v := \Sigma\{agg_{v'} \mid v' \in n_v(t) \wedge par_{v'} = v\} + 1$.
- par_v is some $v' \in n_v(t)$ with minimum level. If $n_v(t) = \varnothing$ then $par_v = v$.

Upon joining, a node v updates its level and parent registers as above, and initializes its aggregate register to 1. We say that a node v *j*oins at level $x \in \omega$, if, upon completed initialization, $lvl_v = x \neq \infty$.

The root node updates only its aggregate register, as above, and initializes lvl_{v_0} to 0 and par_{v_0} to v_0.

It is not hard to see that in the static case, this algorithm converges in expected time λ_g^{-1} times the diameter of the network. In fact, the algorithm is self-stabilizing [40] in the sense that convergence is independent of how the registers are initialized.

Our goal is to characterize the expectation value of the following quantities:

- $N_x(t)$: The number of nodes $v \in V(t)$ with $lvl_v(t) = x$.
- $A_x(t)$: The sum of local aggregates $agg_v(t)$ for nodes $v \in V(t)$ with $lvl_v(t) = x$.

In order to realize this we find that a number of auxiliary quantities also need to be characterized, namely:

- $N_x^{x'}(t)$: The number of nodes $v \in V(t)$ with level x and minimum neighbour level x'.
- $N_x^s(t)$: The number of nodes $v \in V(t)$ with level x that are *s*table in the sense that either $x = 0$ or the minimum neighbour level is $x - 1$. $N_x(t) - N_x^s(t)$ is then the instantaneous number of *u*nstable nodes at level x.

10.4 A theory of GAP under churn

We wish to analyze the performance of the protocol defined in the previous section, for the case of constant churn. This is a complicated task, since every single join or fail event causes an avalanche of restructuring events on the tree overlay. The system is continually in flux due to incessant joins and failures. As a result, for our investigation, we expect the system to be in a steady state, in which every level has a time-independent expected number of stable as well as unstable nodes. In this regime, we analyze GAP by modeling it as a continuous-time Markov process.

For our purposes the state of the system is given by the vector

$$\mathbf{N}(t) \equiv \{\mathbf{N}_0(t), \mathbf{N}_1(t), \mathbf{N}_2(t), \mathbf{N}_3(t), \cdots\} \tag{10.1}$$

where each $\mathbf{N}_i(t)$ is $\left(N_i^s(t), N_i^{i'}(t)_{i' \neq i-1}\right)$. $N_i^s(t) = N_i^{i-1}$ is the instantaneous number of stable nodes at level i at time t and $N_i^{i'}(t)$ is the instantaneous number of nodes at level i whose best connection (i.e., neighboring node with minimal level) is to a node at level $i' \neq i - 1$. The total number of unstable nodes which is also the cardinality of the set $\mathbf{N}_i^{us}$ at level i is then simply $\mathbf{N}_i^{us} = \sum_{i' \neq i-1} N_i^{i'}$.

Note that 'stable' and 'unstable' nodes at level i, as defined above, are simply those nodes which instantaneously either have their best connection at level $i - 1$ (stable) or not (unstable). A more accurate description of the system might hence entail not only keeping track of whether a node is stable or unstable, but also keeping track of the state of a node's connections. For the purpose of making precise estimates for the expected values of quantities of interest, however, we will see that our level of description suffices.

The equation for the time-evolution of the probability distribution (the master equation, in the terminology of [113]) may be written in the usual way as

$$\begin{aligned} \frac{\partial}{\partial t} P\left(\mathbf{N}_1(t), \mathbf{N}_2(t), ...\right) &= \sum_{\mathbf{N}'} w(\mathbf{N}'(t), \mathbf{N}(t)) P(\mathbf{N}'(t)) \\ &- w(\mathbf{N}(t), \mathbf{N}'(t)) P(\mathbf{N}) \end{aligned} \qquad (10.2)$$

Here $w(\mathbf{N}(t), \mathbf{N'}(t))$ is the rate at which transitions occur from the state $\mathbf{N}(t)$ to the state $\mathbf{N}'(t)$ (these terms are called loss terms) and $w(\mathbf{N'}(t), \mathbf{N}(t))$ is the rate at which transitions occur from the state $\mathbf{N}'(t)$ to the state $\mathbf{N}(t)$ (these terms are called gain terms).

If we are able to correctly estimate these transition rates for the GAP protocol, the description of GAP in terms of a master equation would be exact.

From the GAP protocol, the join and fail rates are easy to estimate exactly. A join at level x is a Poisson process (by definition) with rate $\lambda_j p_{min}(x-1)$ where

$$p_{min}(x-1)(t) = \sum_k Q(k)[(1 - \sum_{i=0}^{x-2} \frac{N_i(t)}{N})^k - (1 - \sum_{i=0}^{x-1} \frac{N_i(t)}{N})^k] \qquad (10.3)$$

This expression is simply the probability (accurate for large N) that at least one of the nodes' k links is connected to a node at level $x-1$ while no link has a connection to a lower level. This quantity is averaged over k, and the distribution $Q(k)$ is the *a* priori degree distribution, which is in our case a Poisson distribution.

Failures of nodes occur independently of each other at rate λ_f.

While the rates of individual join and failure events are simple to quantify, the effect that joins and failures have on the system are less easy to quantify. A single join at level x increases the number of stable nodes in this level by 1, but causes a number of 'internal' transitions at several higher levels. (By internal transition, we mean a transfer of nodes from stable to unstable, or unstable to stable, at the same level. These transitions do not change the total number of nodes at a level). For example, if a node joins level 1, *any* other node it connects to in the network which does not already have level 1 or 2, will immediately become unstable by definition, causing an internal transition at that level.

Similarly a single failure at level x causes an internal transition at level $x+1$, since all the nodes which had only this node as their connection to level x become unstable. (Let us call the number of these nodes $\overline{D}_x$.)

A join process hence gives rise to gain (and corresponding loss terms) in the master equations for each level x which look like

$$\begin{aligned} P \quad & (\mathbf{N}_1, \mathbf{N}_2 \cdots, \{N_x^s - 1, \mathbf{N}_x^{us}\}, \cdots \{N_i^s + d_i, \mathbf{N}_i^{us} - d_i\}, \\ , \quad & \cdots, \{N_j^s + d_j, \mathbf{N}_j^{us} - d_j\}, \cdots) \lambda_j Q(x, d_i, d_j ...) \end{aligned} \qquad (10.4)$$

The term above indicates that a join at level x of a node which has d_i connections to level i, d_j connections to level j *e*tc causes $\sum_i d_i$ nodes at a set of levels $\{i, j, \cdots\}$ (each level in this set is necessarily $> x + 1$) to transfer from stable to unstable. Note that $\mathbf{N}_i^{us} + 1 \equiv \{N_i^1, N_i^2, .., N_i^x + 1..\}$ here. The probability distribution Q can be calculated from the a-priori degree distribution $Q(k)$, in much the same manner as $p_{min}(x-1)$ is calculated in Equation 10.3, by calculating the probability that a new joinee at level x, with degree k, has at least one connection to level $x-1$ and d_i connections to level i, d_j connections to level j *e*tc.

The master equation will consist of gain and loss terms of the above sort, summed over x, all allowed sets $\{i, j \cdots\}$ for each x and for each set $\{i, j, \cdots\}$, all allowed values of $\{d_i\}$.

Similarly, the gain and loss terms for each fail event, as well as each GAP update event, will look similar, with the added complication that now we are not justified in using the a-priori distribution to estimate the probability of a failing or transferring node to have d_i connections to level i (and d_j connections to level j *e*tc). The reason the a-priori distribution is not justified in this case is because a node at level x which has spent some time in the network has a different probability to be connected to other levels than a new joinee(*e*.g. if the number of update events per join or fail event is very large, a node at level x would only be connected to levels $x-1$, x and $x+1$, as opposed to a new joinee).

Working with equation 10.2 is hence untenable at this level of detail. However if we are interested in $E[N_x]$, one can show that the above master equation gives the following *e*xact rate equation

$$\begin{aligned}\frac{\partial E\left[N_x\right]}{\partial t} &= -\lambda_f E[N_x] + \lambda_j p_{min}(x-1) \\ &+ \lambda_g \left(\Sigma_{x' \neq x} E[N_{x'}^{x-1}] - E[N_x^{us}]\right)\end{aligned} \tag{10.5}$$

If we are only interested in the steady state (as in the rest of this paper), then we can scale time by λ_f and use the steady state condition $\lambda_j = E[N]\lambda_f$ to get

$$\frac{E[N_x]}{E[N]} = p_{min}(x-1) + r\left(\Sigma_{x' \neq x} \frac{E[N_{x'}^{x-1}]}{E[N]} - \frac{E[N_x^{us}]}{E[N]}\right) \tag{10.6}$$

where $r = \lambda_g/\lambda_f$. The above equation can also be obtained by the following (more transparent, though equivalent) procedure of estimating the rates of increase or decrease of the instantaneous quantity N_x. These rates are determined as follows:

$$N_x(t+\Delta t) \rightarrow \begin{cases} N_x(t)+1 & \text{w.p. } \lambda_j p_{min}(x-1)(t) \\ N_x(t)-1 & \text{w.p. } \lambda_f N_x(t) \\ N_x(t)+1 & \text{w.p. } \lambda_g \sum N_{x'}^{x-1}(t) \; x' \neq x \\ N_x(t)-1 & \text{w.p. } \lambda_g N_x^{us}(t) \end{cases} \tag{10.7}$$

The first (gain) term quantifies the change when a node joins and the second (loss) term accounts for node failures. The third term is the probability that a node with an incorrect level assignment actually has its lowest connection at level $x - 1$ and so is liable to join level x upon an update. This constitutes an influx into level x due to node updates. The last term quantifies the probability that a node with incorrect level x updates its level and is removed from level x. This term thus estimates the outflux from level x due to node updates.

Equation 10.6 is exact with the caveat that in its derivation, the probability of disconnection from the root partition has been ignored. Disconnection may happen for several reasons. A node may join with no connections (the probability for this is simply $exp(-\overline{\lambda})$), or the failure of a node might cause a connected cluster of nodes to disconnect from the main partition. The latter probability also decreases exponentially with $\overline{\lambda}$ for Erdös-Rényi graphs.[1]

The term inside the brackets in Equation 10.6 is the difference of the *i*nflux of nodes after an update event into level x, from an *o*utflux of unstable nodes out of x. On general grounds, this term may be expanded in a series $\sum y_k/r^k$. The first term in the sum which is of order $1/r$, when multiplied by r will be independent of r and hence to leading order, we expect N_x to be independent of r. However if the term inside the brackets is non-zero, we would have sub-leading order corrections, and hence an r-dependence. In the following, we show that for large r ($r >> 1$) this does not happen by proving the following Lemma:

Lemma 10.4.1 *In steady state,for large N, large r and large $\overline{\lambda}$, the expected influx of nodes into level x, $\sum_{x' \neq x} E[N_{x'}^{x-1}]$, is equal to the expected outflux of nodes from level x, $E[N_x^{us}]$.*

Note first that there is no obvious reason why such a balance condition should hold for our system. This is not a requirement for a steady state, nor is it a special condition like detailed balance which holds for reversible dynamics.

We prove this Lemma by proving it for each level x. For level 1, the Lemma is trivially satisfied since both influx and outflux are identically 0 at every instant. The average occupation of level 1 is hence independent of r.

In addition, the expected number of children per level-1 node (*i*.e., the connections to levels other than the root and other level-1 nodes) is also independent of r, using the same analogy with birth-death processes as referred to in proposition 10.2.1. Namely, the time-evolution of any specific level-1 node's connections to higher levels may be construed as a branching process with birth rate $\lambda_j p_{min}(1)/N_1$ (the rate at which a node joins level 2 and connects to a specific node at level 1) and death rate $\lambda_f k$ (for a node with degree k) and hence a Poisson distribution with an r-independent parameter. Since all connections of a level-1 node to higher levels are either to nodes at level 2 or to nodes at levels other than 2, this implies

[1]This might be simply estimated as follows. A node gets disconnected if all k of its neighbours get disconnected. The probability of disconnection P_d must hence satisfy the self-consistent equation $P_d = \sum_k Q(k) P_d^k$ for large enough graphs. When $Q(k)$ is the Poisson distribution, this equation becomes $P_d = exp(\overline{\lambda}(P_d - 1))$ which for $\overline{\lambda} > 4$ gives $P_d \sim exp(-\overline{\lambda})$.

$$\frac{E[N_2^s]}{N} + \sum_{x' \neq 2} \frac{E[N_{x'}^1]}{N} = \frac{E[N_2]}{N} + \left(\Sigma_{x' \neq 2} \frac{E[N_{x'}^1]}{N} - \frac{E[N_2^{us}]}{N} \right) \tag{10.8}$$

is independent of r. Here (and in all that follows), we have replaced $E[N]$ by N for ease of notation.

However from Equation 10.6 we know that

$$\frac{N_2}{N} = p_{min}(1) + r \left(\Sigma_{x' \neq 2} \frac{N_{x'}^1}{N} - \frac{N_2^{us}}{N} \right) \tag{10.9}$$

which is strictly true for a system where we can ignore partitioning. Using equations 10.8 and 10.9, we see that the expression

$$p_{min}(1) + (r+1) \left(\Sigma_{x' \neq 2} \frac{N_{x'}^1}{N} - \frac{N_2^{us}}{N} \right) \tag{10.10}$$

is independent of r. Since $p_{min}(1)$ is independent of r, the only way the above expression can be independent of r is if the term in the bracket $= 0$. This proves the Lemma for level 2 for any r. The above procedure is, however, hard to generalize for higher levels. So, we provide an alternative proof of Lemma 10.4.1 for level 2 which holds strictly only for large r. Consider now a system in steady state, where we switch off joins and failures at all levels except at level 1.

For such a system the steady state condition for level 2 implies

$$\Sigma_{x' \neq 2} \frac{E[N_{x';2}^1]}{N} = \frac{E[N_{2;2}^{us}]}{N} \tag{10.11}$$

where $;2$ in the subscript indicates that the influx and outflux into level 2 have been estimated with joins and fails only at levels lower than level 2 (hence level 1).

But, by the GAP protocol, transitions to and from level x are $o$nly caused by joins and fails at lower levels. Hence, if we set out to elaborate gain and loss terms for N_2^{us} (in the way it is done for N_x in Eq. 10.7), then $a$ll gain terms (i.e., terms which lead to an increase in N_2^{us}) are entirely unaffected by joins and failures at levels 2 and higher. Failures at level 2 give rise to an extra loss term $-\lambda_f N_2^{us}$, which, however, is much smaller than the loss term $-\lambda_g N_2^{us}$ for large r. Hence for large r

$$\Sigma_{x' \neq 2} \frac{E[N_{x';2}^1]}{N} = \Sigma_{x' \neq 2} \frac{E[N_{x'}^1]}{N} \tag{10.12}$$

and

$$\frac{E[N_{2;2}^{us}]}{N} = \frac{E[N_2^{us}]}{N} \tag{10.13}$$

The above argument shows that the expressions $\frac{E[N_2^{us}]}{N}$ and $\Sigma_{x' \neq 2} \frac{E[N_{x'}^1]}{N}$ can be calculated accurately to order $1/r$ by considering a system with joins and fails only at level 1.

We now prove Lemma 10.4.1 for an arbitrary level x, by generalising the above argument and considering a steady state where all joins and failures at levels x and higher have been switched off.

In this case, for level x, the steady state condition gives

$$\Sigma_{x' \neq x} \frac{E[N^{x-1}_{x';x}]}{N} = \frac{E[N^{us}_{x;x}]}{N} \tag{10.14}$$

where $;x$ in the subscript indicates that the influx and outflux into level x have been estimated only by considering joins and failures at levels lower then level x. If we now introduce churn at levels x and higher levels, the same argument as we used for level 2 indicates that the 'true' expression $\frac{E[N^{us}_{x}]}{N}$ differs from the expression $\frac{E[N^{us}_{x;x}]}{N}$ only to order $1/r^2$. Similarly the expression $\Sigma_{x' \neq x} \frac{E[N^{x-1}_{x'}]}{N}$ calculated when there is churn at all levels, differs from $\Sigma_{x' \neq x} \frac{E[N^{x-1}_{x';x}]}{N}$ only to order $1/r^2$.[2]

In fact, by dividing influx and outflux at any level x into components from higher and lower levels, it might be possible to prove a stronger version of Lemma 10.4.1, implying two separate balance conditions. More details of the gain and loss terms involved in a calculation of N^{us}_x are given in the appendix.

Note that our proof of the Lemma does not rely on any graph property, and so we expect the Lemma to hold in this form, even for graphs that are not Erdös-Rényi graphs. The validity of Lemma 10.4.1 implies, from Eq. 10.7, that

$$E[N_x]/N = p_{min}(x-1) \tag{10.15}$$

Thus if influx and outflux are in equilibrium, the level distribution is independent of r and equal to the a priori distribution.

Indeed this is confirmed by the simulation results in Figure 10.1, with caveats related to the possibility of nodes disconnecting from the root (which as we have mentioned is exponentially small in average node degree and hence is neglected in our estimations.)

We have proved Lemma 10.4.1 for large r and hence the result $E[N_x]/N = p_{min}(x-1)$ would seem to hold only for large r. However, it is trivial to show for this protocol that the very same result also holds for $r = 0$. Note though that large r and $r = 0$ having the same steady state value of $E[N_x]/N$ does not rule out a different behavior for intermediate values of r.

In what follows, we make use of Lemma 10.4.1 to estimate the expected value of the subnetwork rooted at a node at level x (next section), as well as the number of unstable nodes at level x (the details of which are in the appendix).

[2] This term considers the influx into level x in steady state and hence can be divided logically into two parts, influx from lower levels and influx from higher levels. The influx from lower levels is not affected at all by churn at levels x and higher. Each influx from a higher level x is however modified by $O(1/r^2)$ when churn is allowed at all levels.

10.5 Aggregation

We now turn to the aggregation part of the protocol running on top of the induced tree overlay. As described in Section 10.3, a node updates its agg_v register by aggregating the value of its childrens agg registers and adding one to count itself. The value agg_v will then, in a static network, clearly converge to the size of the sub-tree rooted at node v and agg_{v0} will give the total network size. In order to calculate the average estimation error made by a node at level x as a function of churn r, we introduce the following variables:

Let $E[M_x(t)]$ be the average value of the aggregate held by a node at level x at time t and let $E[A_x(t)]$ denote the average value of the sum of the aggregates held by the nodes at level x at time t ($E[A_x(t)] = E[M_x(t)N_x(t)]$).

We keep track of the gain and loss terms that lead to an increase or decrease in the instantaneous value $A_x(t)$ in much the same way as in Equation 10.7 to get

$$A_x(t+\Delta t) \rightarrow \begin{cases} A_x(t) - M_x(t) & \text{w.p. } \lambda_f N_x \\ A_x(t) + 1 & \text{w.p. } \lambda_j p_{min}(x-1) \\ A_x(t) - M_x(t) & \text{w.p. } \lambda_g N_x^{us} \\ A_x(t) + M_{x'}(t) & \text{w.p. } \lambda_g \sum N_{x'}^{x-1} \;\; x' \neq x \\ A_x(t) - M_x(t) + 1 + & \\ \;\sum_{i=1}^{D_x} M_{x+1}^i(t) & \text{w.p. } \lambda_g N_x^s \end{cases} \tag{10.16}$$

These terms account for all the processes, which in an infinitesimal unit of time, could change A_x: Level x loses the instantaneous value $M_x(t)$ upon failure of a node at level x; upon a join, A_x is increased by one; upon an update of an unstable node leaving level x, the instantaneous value $M_x(t)$ is lost; upon an update of an unstable node at level x' entering x, $M_{x'}(t)$ is gained; finally, upon an update of a stable node, the current expected aggregate is lost, and the sum of the aggregates of the children (whose instantaneous value is $D(x)$) plus one is gained.

Using $a_x = E[A_x]/N$ the above gain and loss terms give the following exact equation for a_x in the steady state:

$$\begin{aligned} a_x(1+r) &= p_{min}(x-1) + r \sum_{x' \neq x} \frac{E[M_{x'} N_{x'}^{x-1}]}{N} \\ &+ r\frac{E[N_x^s]}{N} + r\frac{E\left[N_x^s \sum_{i=1}^{D_x} M_{x+1}\right]}{N} \end{aligned} \tag{10.17}$$

To make any progress with this equation, we need to simplify the second and last terms as well as estimate $E[N_x^s]/N$. The second term is the sum of the aggregate of all nodes moving into x. To estimate this quantity we make the simplifying assumption (consistent with Lemma 10.4.1), that for $x' \neq x$, $N_{x'}^{x-1}$, the influx to x

from x', is roughly equal to the population at level x' times the ratio of outflux of level x to the entire population, i.e.

$$\Rightarrow E[N_{x'}^{x-1}] \approx \frac{E[N_x^{us}]}{N} E[N_{x'}].^3 \tag{10.18}$$

Note that this assumption can itself be used to estimate N_x^{us} but gives a far worse estimate than the one we derive in the appendix. However, the error made by substituing this into Eq. 10.17 is very small.

The last term we approximate as $E[D_x]E[N_x^s]E[M_{x+1}]$. This approximation is expected to hold as long as the number of children a node has is not correlated to the value of their aggregate.

The number of children is approximated by

$$E[D_x] = E[N_{x+1}^s]/E[N_x] \tag{10.19}$$

This is a good estimate but not exact, as there might be nodes at level $x+1$ which have a better choice of parent but do not as yet know this (not having executed an update). In this case, from the protocol, they would still send their aggregate to their outdated parent despite being officially classified as an unstable node. We ignore this effect in the theory and again the error made in introducing this approximation into Eq. 10.17 is negligible.

Using these approximations and Lemma 10.4.1 Eq. 10.17 becomes

$$\begin{aligned} a_x = p_{min}(x-1) + \tfrac{r}{r+1}(a_{x+1}\tfrac{E[N_x^s]}{E[N_x]}\tfrac{E[N_{x+1}^s]}{E[N_{x+1}]})+ \\ \tfrac{r}{r+1}(\tfrac{E[N_x^{us}]}{N} \sum_{x'\neq 0,1,x} (a_{x'} - p_{min}(x'-1)) \end{aligned} \tag{10.20}$$

This equation can be solved numerically, once we have an expression for $E[N_x^{us}]$ as a function of r. We present this calculation in the appendix. The calculation involves estimating the outflux out of level x due to all the processes which might make a node at level x unstable: namely, the node has a single connection to the earlier level which fails; or the single connection is itself unstable and on an update makes the level-x node unstable; or a new joinee or one of the neighbors of the node provides a better choice of parent. We show from comparison to simulations, that our estimate for $E[N_x^{us}]$ is quite good for all the values of the parameters considered.

We input this estimate of $E[N_x^{us}]$ into Eq. 10.20 above and solve it iteratively for a_x to get a self-consistent solution. In the next section we test our theory against simulations for various values of the parameters r, N and $\overline{\lambda}$

[3] In fact the use of N here is a slight overestimation, as levels 0,1, and x itself need to be excluded.

10.6 Validation Through Simulation

We evaluate the model developed in Sections 10.4 and 10.5 through simulation studies using a discrete-event simulator. We simulate the protocol described in Section 10.3 executing on a network modeled as described in Section 10.2. The events we simulate are nodes joining, nodes failing and the execution of protocol cycles on a node. (The protocol does not distinguish between a node failing and a node leaving the network.)

When a join event occurs, a new node is created on the network graph, links from this node to other nodes of the network graph are created, the node's registers are initialized following the join protocol described in Section 10.3, and the node executes a protocol cycle. When a fail event occurs, the node is removed from the network graph, together with the links that connect it to other nodes in the graph. Finally, during the execution of a protocol cycle on a node, the registers of the node are updated as described in Section 10.3, whereby the node changes its parent only if a neighbor with a lower level (a level closer to the root) than its current parent exists.

During a simulation run, we periodically sample the metrics N, N_x, A_x and N_x^{us}. Each measurement point on the graphs in this section corresponds to values averaged over at least a 1,000 such samples.

All runs simulate the system in steady-state. The network graph for the simulation is a random graph with N nodes witha probability p that any pair of these nodes is connected. ($p = \overline{\lambda}/N$) where $\overline{\lambda}$ is the average node degree.

We perform simulation runs for all combinations of the following parameters: number of nodes (N): 1,000, 10,000, 100,000; average node degree ($\overline{\lambda}$): 4, 6, 8; r: 1, 10, 100, 1,000. Runs for a single simulation can be very long for large networks and small r. For instance, for $N = 100,000$ and $r = 1$, the simulation takes about a month on a single server to collect 1,000 samples.

Using these simulation results, we validate the following predictions of our models, for steady state conditions: the level distribution, the accuracy of the counting protocol and the distribution of unstable nodes. The first two predictions are discussed in this section, while the last is discussed in the appendix.

Validation of Level Distribution ($E[N_x]/N$) In this series of experiments, we validate the prediction for $E[N_x]$ (see Section 10.4) under steady-state conditions.

Specifically, we compare the simulation results with the prediction of equation (10.15).

Figure 10.1 shows in a series of graphs, the numerical evaluation of equation (10.7), together with simulation results, for network sizes of 1,000 and 100,000, degrees of 4 and 8, and rates of 1, 10, 1,000.

We make the following observations. First, the model predictions fit the actual simulation results very well. As predicted, the distribution seems to be independent of the parameter r. Second, as expected, an increase of the network size, or a decrease of the node degree enlarges the average tree height.

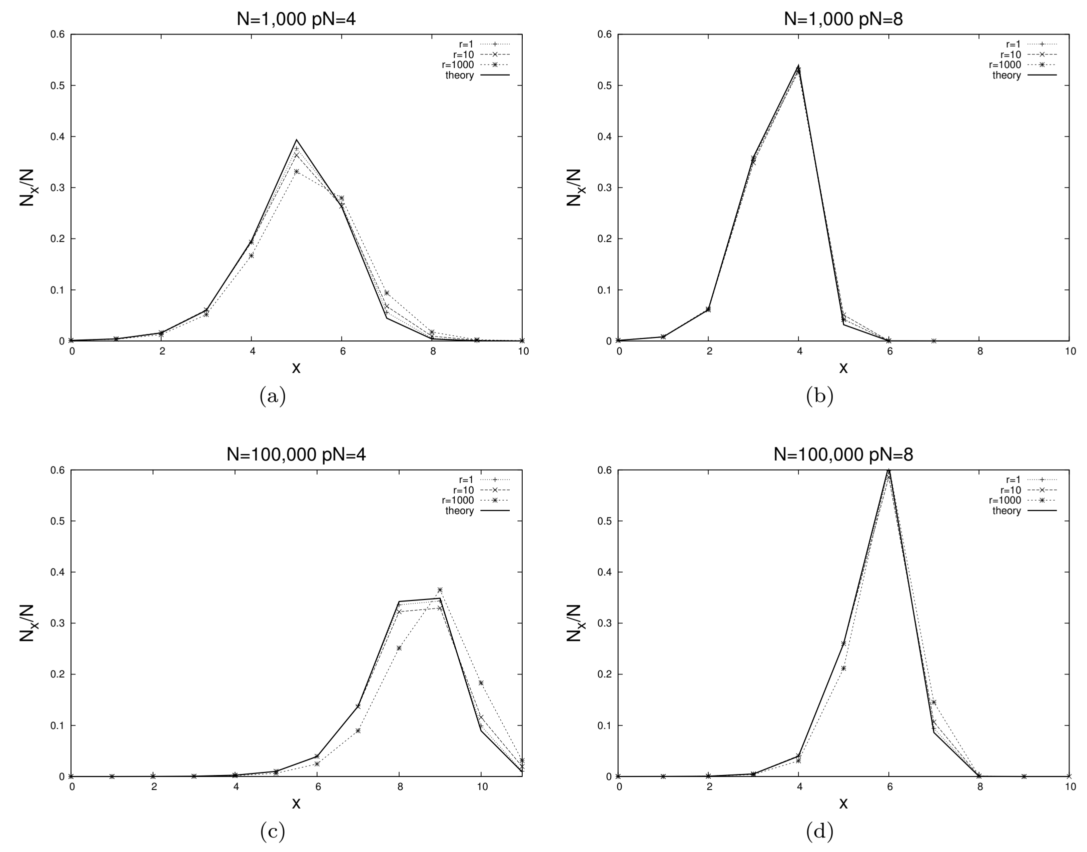


Figure 10.1: Comparing theory with simulation results: the level distribution $E[N_x]/N$ as a function of network size, node degree $\overline{\lambda} = pN$ and r.

Validation of a_x In this series of experiments, we validate our prediction for a_x (see Section 10.5) under steady-state conditions.

Figure 10.2 shows, in a series of graphs, the numerical evaluation of equation (10.20), together with selected simulation results, for network sizes of 1,000, 10,000, 100,000, degrees of 4, 6, 8, and rates of 1, 10, 100, 1,000.

These results suggest the following. First, the accuracy of the prediction of the model increases with r and N, and decreases with x. Second, we observe that the accuracy of the counting protocol decreases with increasing network size and, as expected, the accuracy increases with the degree of the network graph and the parameter r.

Of specific interest to us is the accuracy of the counting protocol, whichdepends

	$r = 1$			$r = 10$		
$\overline{\lambda}$	4	6	8	4	6	8
$N = 10^3$	.03	.05	.07	.02	.00	.01
$N = 10^4$	.12	.27	.28	.11	.00	.04
$N = 10^5$	.37	.56	.65	.58	.10	.00

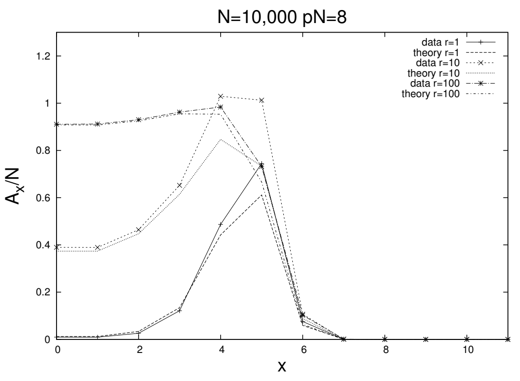
	$r = 100$			$r = 1,000$		
$\overline{\lambda}$	4	6	8	4	6	8
$N = 10^3$	.02	.00	.01	.03	.00	.00
$N = 10^4$	.03	.01	.00	.09	.00	.00
$N = 10^5$	.16	.02	.00	.04	.00	.00

Table 10.1: Relative errors of model prediction of a_0 versus simulation results in steady state, for network size N, average node degree $\overline{\lambda} = Np$ and $r = \lambda_g/\lambda_f$.

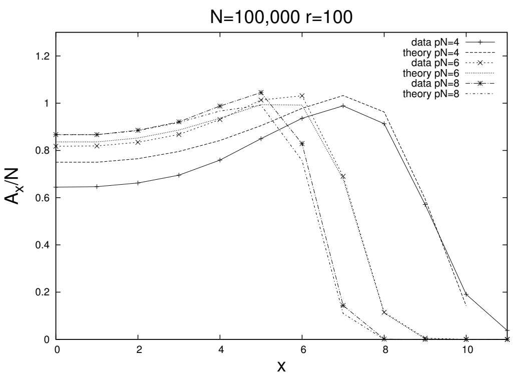
on the prediction accuracy of a_0 (the value at the root). Table 10.1 gives the relative errors of the model prediction of a_0 against the simulation measurements for various parameters. We notice that the predictions of the model are good for large values of r. For instance, for $r = 100$, i.e., for systems with an average lifetime of about 100 protocol cycles per node, the prediction error is at most 2%, for network graphs with average node degree of 6 or 8. We can also see that for a highly dynamic system with $r = 1$, the error is much larger. To illustrate the system dynamics, in the scenario with $r = 1$, $\overline{\lambda} = 8$, and $N = 100,000$, the model predicts $a_0 = 201$, while the simulation gives $a_0 = 122$, which is hopelessly wrong for a counting protocol.

10.7 Discussion

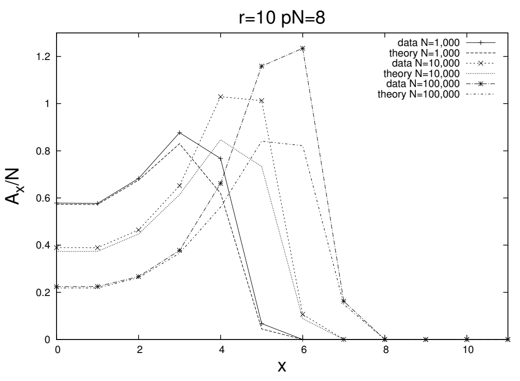
Our procedure of describing the GAP protocol, in the regime of incessant churn, by a continuous-time Markov process has many analogies to the modelling and performance analysis of communication networks using queueing networks [63]. In fact we can think of each level in our system as an $M/M/\infty$ queue, with Poisson input, and a protocol-determined rule for how 'customers' transfer between queues, or leave the system, once they are serviced. At the simplest level of such a description, the GAP protocol may be simply viewed as an open migration process [63] with a queueing node i (level i) consisting of N_i customers. In such a description, customers enter the queue in question as a Poisson process and leave independently of each other (as they also do in the continuous time Markov description of GAP). However, the analytical intractability of the continuous-time Markov process, arising from the simultaneous internal transitions caused in several 'queues' by a single join, fail, or transfer event is done away with in the open migration process by assigning an intrinsic transfer rate λ_{ij} to each node to transfer from queue i to queue



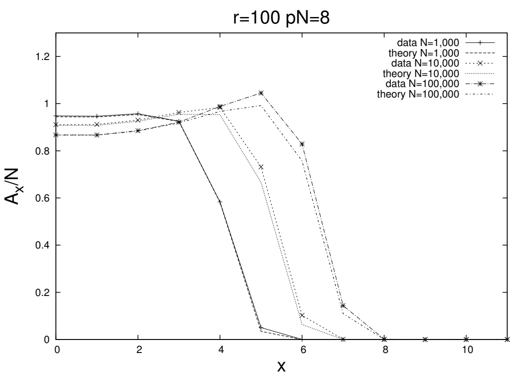
(a)



(b)



(c)



(d)

Figure 10.2: Comparing theory with simulation results: a_x as a function of network size, node degree and r. For $x = 0$, the curves show the accuracy of the protocol for counting.

j. Interestingly, λ_{ij} may be chosen so as to give the same equation for expected values, namely Equation 10.6, as a global balance condition. Such a migration process has a product form steady state solution, as investigated for queuing networks by Kelly and others, and obeys in particular Kelly's partial balance conditions [63]. Its an interesting and open question whether the continuous-time Markov version of GAP also has a product form solution despite its more complicated description.

An interesting aspect of our work which gives rise to many open questions is Lemma 10.4.1. This balance condition does not come out in our analysis as a general property of quasi-reversible queueing networks (aka Kelly) but seems to arise in GAP because of the details of the protocol. Could one classify what rules a queueing network would have to obey so that such a Lemma would hold? We see in our analysis that the existence of such a balance condition provides an invaluable tool for simplifying the analysis. It would be interesting if other tree-based protocols also showed similar behaviour. In addition, investigating whether Lemma 10.4.1 holds for any r is also an open and important question.

Another relevant question in talking of tree-based protocols is how the underlying graph affects the performance of the protocol. In our case, we have chosen to work with an Erdös-Rényi graph, and the graph properties (namely the Poisson degree distribution) play a role in the estimation of all the quantities of interest. The proof of Lemma 10.4.1 however does not make any reference to the nature of the underlying graph. Even in the queueing network analogy, the existence of a product-form steady state does not depend on graph properties, but rather only on the λ_{ij}'s being such as to allow customers to reach any queue from any other. It would thus be interesting to tease out which properties depend on details of the underlying graph and which do not and whether the very same protocol with the same parameters would function better on some graphs more than others.

10.8 Summary

The analysis presented in this paper, is not rigorous. Most of the results only hold for large N (in which regime Equation 10.3 is accurate), large $\overline{\lambda}$ (when partitioning effects can be ignored) and large r (which is the regime in which we have shown that Lemma 10.4.1 holds, though it may possibly also hold for any r). However these are the interesting regimes to look at since these correspond to large networks, underlying graphs with a very small disconnection probability and a large maintenance rate. As expected (see Section 10.6) numerical experiments agree better with our theory as N,r and $\overline{\lambda}$ increase. In addition a comparison between data and theory shows (see Table 1 for instance) that our predictions are reasonable even for $r = 10$ when $\overline{\lambda} > 4$ and $N \sim 1000$.

In summary the contribution of this paper is a first step towards a general theory of tree-based protocols under churn. We have shown that key performance metrics of a simple, tree-based aggregation protocol can be calculated systematically, and that these calculations are mostly in good agreement with simulation results. In addition, the equations we have derived can be solved numerically for any set of parameter values, to predict the expected accuracy of the estimate at the root, which requires only a few seconds on a regular laptop. (This is in contrast to a simulation, which, depending on the parameter values of interest, can take months on a single processor.) We believe that using such analytical tools to predict performance metrics can be of invaluable assistance in understanding the behavior of large-scale distributed information systems, and hence open up exciting avenues for engineering new protocols that execute effectively in dynamic environments.

.1 Estimation of $E[N_x^{us}]$

We present below the details of the calculation for $E[N_x^{us}]$. This calculation goes through the usual steps of detailing gain/loss terms, finding appropriate expressions for these and solving the equation for the expectation value in the steady state. As in Sections 10.4 and 10.5, we make use of both proposition 10.2.1 and Lemma 10.4.1. Also, as in Section 10.5, we make the approximation that we can estimate $E[XY]$ by $E[X]E[Y]$. In addition, we use the a-priori distribution to estimate some quantities related to the degree of a node at level x in the network, even though, as we have remarked, the a-priori distribution is strictly only correct for new joinees. One justification for this procedure is that the steady-state value of some (though by no means all) quantities for this system seem to be the same for small r as well as large r as remarked upon in Section 10.4. For small r, since there are hardly any node-exchanges between any two levels per node lifetime, clearly the a-priori distribution works. Hence it must do so for large r as well. A stricter justification would however entail proving that the large r and small r- limits are the same, and that the error made in using this distribution even for intermediate r is bounded, but we do not do that here.

The $x \geq 2$ case Since the root never fails, nodes at level 1 can never become unstable. We thus need to estimate unstable nodes only for level $x \geq 2$ We obtain the following gain-loss terms for the instantaneous quantity $N_x^{us}(t)$:

$$N_x^{us}(t+\Delta t) \rightarrow \begin{cases} N_x^{us}(t) - 1 & \text{w.p. } \lambda_f N_x^{us}(t) \\ N_x^{us}(t) - 1 & \text{w.p. } \lambda_g N_x^{us}(t) \\ N_x^{us}(t) + 1 & \text{w.p. } (A) \\ N_x^{us}(t) + 1 & \text{w.p. } (B) \\ N_x^{us}(t) + 1 & \text{w.p. } (C) \end{cases}$$

The probabilities (A)–(C) are determined as follows:

(A) This is the probability that a single connection at level $x-1$ fails or becomes unstable. If the connection fails, by definition the node at level x automatically becomes unstable. If the connection becomes unstable, the level-x node becomes unstable *o*nly after the parent has performed an update, which explains why the second term in the following expression is multiplied by r. Hence this term (A) should be $\lambda_f(N_{x-1}\overline{D}_{x-1} + rN_{x-1}^{us}\overline{D}_{x-1})$ where we have used $\lambda_g = r\lambda_f$ and we have introduced the notation $\overline{D}_{x-1})$ to denote those children of a node at level $x-1$ which only have this node as their connection to level $x-1$. Since the final equation we are interested in is for $E[N_x^{us}]/N$, we need to estimate $(A) = E[(N_{x-1}\overline{D}_{x-1}]/N + rE[N_{x-1}^{us}\overline{D}_{x-1})]/N$.

$E[N_{x-1}\overline{D}_{x-1}]/N$ is the expected fraction of the number of *s*table nodes at level x which have only one connection to the level $x-1$. We approximate this by $\alpha_x E[N_x^s]/N$ where α_x is the probability of having only one connection to the level above using the a-priori distribution:

$$\alpha_x = \overline{\lambda}\frac{E[N_{x-1}]}{E[N_x]} e^{-\overline{\lambda}\sum_{i=0}^{x-1}\frac{E[N_i]}{N}} . \tag{21}$$

In making this approximation, we are assuming that the quantity $E[N_{x-1}\overline{D}_{x-1}]/N$ is well approximated by the product of two quantities: the fraction of stable nodes at level x times the a-priori probability of a node at level x to have only one connection to the parent level.

Similarly the term $rE[N^{us}_{x-1}\overline{D}_{x-1})]/N$ is approximated by $rE[N_{x-1}\overline{D}_{x-1}/N]\frac{E[N^{us}_{x-1}]}{E[N_{x-1}]} \sim r\alpha_x\frac{E[N^s_x]}{N}\frac{E[N^{us}_{x-1}]}{E[N_{x-1}]}$

We can hence write (A) as:

$$\begin{aligned}(A) &= \alpha_x\frac{E[N^s_x]}{N}\left[1 + r\frac{E[N^{us}_{x-1}]}{E[N_{x-1}]}\right] \\ &= \alpha_x\frac{E[N^s_x]}{E[N_x]}\frac{E[N_x]}{N}\left[1 + r\frac{E[N^{us}_{x-1}]}{E[N_{x-1}]}\right] \end{aligned} \tag{22}$$

(B) This is the probability that a new joinee with a lower level than the parents', connects to a stable node at level x. If such an event occurs, then the node at x has a better choice of parent then the current one at $x-1$, and hence becomes unstable. Using the a-priori distribution, (B) is easily estimated as

$$\begin{aligned}&\frac{E[N^s_x]}{N}\left[\frac{\overline{\lambda}^2}{N}e^{-\overline{\lambda}/N} + \overline{\lambda}\sum_{m=1}^{x-3}[e^{-\overline{\lambda}\sum_{i=0}^{m-1}\frac{E[N_i]}{N}} - e^{-\overline{\lambda}\sum_{i=0}^{m}\frac{E[N_i]}{N}}]\right] \\ &= \frac{E[N^s_x]}{N}\left[\frac{\overline{\lambda}^2}{N}e^{-\overline{\lambda}/N} + \overline{\lambda}[e^{-\overline{\lambda}/N} - e^{-\overline{\lambda}\sum_{i=0}^{x-3}\frac{E[N_i]}{N}}]\right]\end{aligned}$$

This is the average number of connections to stable nodes at level x that a new joinee with a parent at level 0,1 ... up to $x-3$, has. Such a new joinee will then make the node at level x change its level, when it updates.

(C) The probability that a neighbor of a node at level x provides a better choice of parent, i.e. that at least one of the neighbor nodes is unstable with a better parent at level strictly less than $x-2$. When such a neighboring node performs an update, the node at level x becomes unstable.

We first estimate λ_x, the average degree of a node at level x using the a priori distribution. In order to not overcount the case of a single parent (case (A)), we consider *a*ll neighbours of a node with more than one connection to the parent level and all neighbours *e*xcluding the parent, for a node with only one connection to the parent level . λ_x then takes the value

$$\begin{aligned}\lambda_x = & \frac{\overline{\lambda}(1-\sum_{i=0}^{x-2}\frac{E[N_i]}{N})(e^{-\overline{\lambda}\sum_{i=0}^{x-2}\frac{E[N_i]}{N}})}{p_{min}(x-1)} \\ & -\frac{\overline{\lambda}(1-\sum_{i=0}^{x-1}\frac{E[N_i]}{N})(e^{-\overline{\lambda}\sum_{i=0}^{x-1}\frac{E[N_i]}{N}})}{p_{min}(x-1)} \\ & -\frac{\overline{\lambda}\frac{E[N_{i-1}]}{N}e^{-\overline{\lambda}\sum_{i=0}^{x-1}E[N_i]/N}}{p_{min}(x-1)}\end{aligned}$$

Secondly, we estimate the probability of a neighbouring node being unstable with a parent at level $1, \cdots, x-3$. We bound this probability by the sum $\sum_{i=2,x-2}\frac{E[N_i^{us}]}{N}$. This approximation is the same as is made in Eq. 10.18. As a consequence we get (C) $= r\lambda_x\sum_{i=2}^{x-2}\frac{E[N_i^{us}]}{N}$

Putting the above together we obtain:

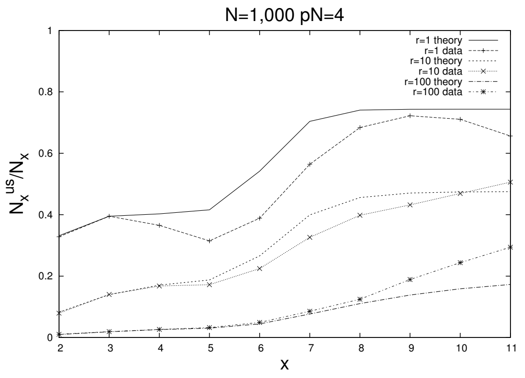
$$\frac{E[N_x^{us}]}{E[N_x]} = \frac{D+E+F}{1+D+E} \tag{23}$$

where

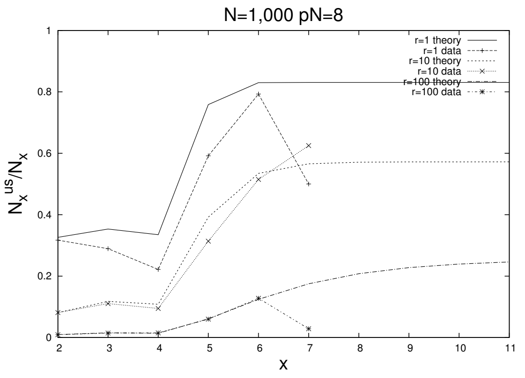
$$\begin{aligned}D &= \frac{\alpha_x}{1+r}(1+r\frac{E[N_{x-1}^{us}]}{E[N_{x-1}]}) \\ E &= [\overline{\lambda}^2 e^{-\overline{\lambda}/N}+\overline{\lambda}(e^{\overline{\lambda}/N}-e^{-\overline{\lambda}\sum_{i=0}^{x-3}\frac{E[N_i]}{N}}]\frac{1}{1+r} \\ F &= \frac{r}{1+r}\lambda_x\sum_{i=2}^{x-2}\frac{E[N_i^{us}]}{N}\end{aligned}$$

Note that Equation 23 is a recursive equation for the quantity $\frac{E[N_x^{us}]}{E[N_x]}$. We hence solve the equation by solving first for $x = 2$ and then recursively for higher levels. Figure .3 shows in a series of plots, the numerical evaluation of Equation 23, together with simulation results, for network sizes network sizes of 1,000 and 100,000, degrees of 4 and 8, and rates of 1, 10 and 1,000.

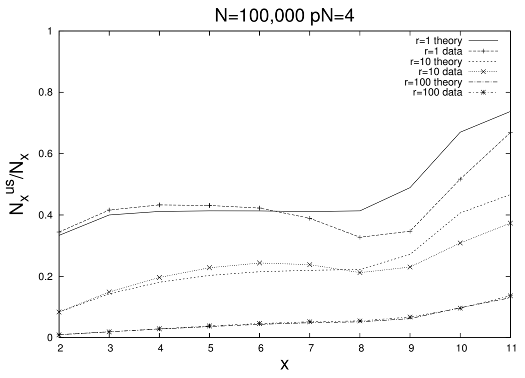
We see that equation 23 predicts simulation results quite well, including the non-monotonic trend in N_x^{us}/N_x as x increases as well as the decrease in this quantity with increasing r.



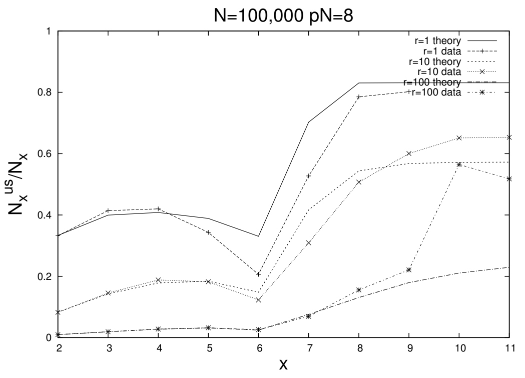
(a)



(b)



(c)



(d)

Figure .3: Comparing theory with simulation results: the fraction of unstable nodes at level x : $E[N_x^{us}]/E[N_x]$ as a function of network size, node degree and r.

Chapter 11

Manycore Work Stealing

Karl-Filip Faxén[1], John Ardelius[1],

[1] Swedish Institute of Computer Science (SICS), Sweden
`[kff,john]@sics.se`

Abstract

Manycore processors comprised of many relatively simple cores interconnected with a switched network form one of the main approaches to future massively parallel chip multiprocessors. On the programming side, nested task parallelism implemented by work stealing schedulers is gaining popularity as a programming model. This paper investigates the convergence of the trends; executing task based programs on a 64 core Tilera processor under the high performance work stealer Wool. We discuss the changes needed to Wool when porting it to the Tilera processor in order to take advantage of its unique features. We measure the performance of (Wool translations of) several programs from the BOTS benchmark suite, as well as microbenchmarks exhibiting extremely fine grained tasks, observing excellent scalability (sometimes over 50 on 63 cores) whenever sufficient parallelism exists.

We also look at two issues that arise when scaling work stealing to larger number of cores: First, while random victim selection provides good load balancing, the number of steals needed and their attendant overhead can be reduced by sampling a few workers and stealing the (heuristically) largest task on offer. Second, for machines where the number of potential victims (ie cores) is large compared to cache and TLB sizes, thieves can benefit from attempting steals from only a subset of the potential victims if multiple thieves are active at the same time.

11.1 Introduction

As manycore processors, which trade the absolute performance of individual cores for a larger number of cores, are becomming available, the question is how to program them. Do the techniqes that have been proposed for multicores scale up, or are entirely new techniques needed? We address this question by porting Wool, a work stealing implementation of the *nested task parallel* programming model, to the TILEPro64, a recent commercial manycore processor. We measure the performance and scaling of a set of programs from the BOTS benchmark suite, and some microbenchmarks used in previous work on Wool. We also make a preliminary exploration of some issues that arise when scaling fine grained work stealing to a 64 core system. In particular, we explore alternatives to purely random work stealing and extrapolate their bahaviour to larger systems using a simplified simulation model.

Task parallelism

In the world of multicores, nested task parallelism has gained a lot of attention [3, 78, 99, 77]. In this explicitly parallel programming model programs create parallelism by *spawning* tasks with which they later *join* (wait for their completion). Task parallelism is often implemented using *work stealing* where a spawn adds work to a local *task pool*. Idle processors *steal* work from the task pools of randomly selected victims. Join checks if the spawned work has been stolen, in which case it blocks until the stolen work is completed, attempting to find other useful work in the meantime. If the spawned work was not stolen (which is typically the more frequent case), it is performed as part of the join operation by the same processor that spawned it.

In nested task parallel programs the tasks that are executed at run-time form a tree where a task that spawns another task is its parent. In the task pools of the processors, each task represents a subtree of the task tree of the entire program. Executing the task, whether by the owner of the pool or by a thief, unfolds the tree.

Wool [44] is a C library / macro package implementing nested task parallelism by means of work stealing. Wool has unusually small overheads for spawn, join and steal and is therefore capable of efficiently exploiting very fine grained parallelism. It uses a novel work stealing algorithm, the *direct task stack.*

The direct task stack contains task descriptors, rather than pointers to task descriptors, and treats these in a strict LIFO order, simplifying memory management for tasks. The owner spawns and joins at the top end of the stack using the `top` pointer, while thieves steal from the bottom using `bot`. Synchronization among thieves and between thieves and owner uses state in the task descriptors rather than the more conventional approach of comparing the top and bottom pointers, allowing `top` to be local to the owner. Since task descriptors are aligned to cache line boundaries, this allows a thief to steal touching just two cache lines; the one containing `bot` and the one containing the stolen task.

Wool uses *leap frogging* [115] to keep workers busy during blocked joins. In this scheme, a blocked worker is only allowed to steal tasks that are grandchildren of the task it is trying to join with (that is, it may steal subtrees of the task tree rooted in the joined task). This solves problems with excessive stack growth that occurs if a blocked worker steals a task that unfolds to a tree deeper than that of the joined task. It also ensures that when the computation is finished, the join can unblock and resume execution immediately, which would not be possible if the joining worker had stolen a large computation that was still executing when the joined task completed.

Leap frogging is not a perfect solution to the blocking join problem since it only allows stealing of a subset of the work in the system. If the joined task is sequential, it forces the joining worker to be idle. Some systems, like TBB, provide a continuation passing API to avoid the blocking join operation altogether, while Cilk solves it using a *cactus stack* built from explicitly linked, heap allocated activation records.

Manycore processors

Manycore processors are not necessarily scaled up versions of conventional processors, but tend to have a different architecture. For instance, the TILEPro64 processor used in this work features 64 rather simple in-order 3-way VILW cores, each with private split 16Kbyte I and 8KByte D L1 and 64Kbyte unified L2 caches, interconnected with a switched mesh network.

The caches of the TILEPro64 are kept coherent by hardware using a variation of a directory based coherence protocol. Each cache line has a unique home tile which is responsible for the coherence state of the line. Writes always update the copy residing in the home tile.

It is currently unclear which memory model future manycore processors will support with the options ranging from cache coherence as in traditional multiprocessors to explicit local memories as in the Cell processor and many GPGPUs. From a programming perspective, cache coherence is attractive, not least becuase it simplifies the porting of legacy code. However, cache coherence has a couple of drawbacks. First, it comes with a higher hardware cost in terms of chip area as compared to explicit local memories (tags etc). Second, it comes with a time overhead in terms of coherence transactions; with local memories, the communication is under explicit control of the application.

In this paper, we find support for (although not a proof of) the feasability of the cache coherent road to manycore computing by demonstrating the scalability of a work stealing scheduler on a set of simple programs on a cache coherent manycore. We use a processor with a relatively light weight memory hierarchy and coherence protocol which has been shown to scale to 64 (and in more recent models 100) cores.

We make the following contributions:

- We give the first analysis of the unbalancing effect of random work stealing and find that sampling can be used to mitigate this effect for systems of over 100 cores but that smaller systems can only benefit if taking a sample is much cheaper than stealing (section 11.2).
- We show how to adapt a low overhead work stealer to manycores in general and the TILEPro64 in particular, presenting a new work stealing algorithm using only a simple test-and-set operation for mutual exclusion (section 11.3), and a novel *victim set* algorithm, which consistently outperforms random victim selection (section 11.3).
- We compare the performance of Wool to theoretical bounds finding that the practical implementation is most often at or very close to these bounds for a set of common task parallel benchmarks (section 11.4).

11.2 Unbalanced stealing

In nested task parallel programs the tasks that are executed at run-time form a tree where a task that spawns another task is its parent. In the task pools of the processors, each task represents a subtree of the task tree of the entire program. Executing the task, whether by the owner of the pool or by a thief, unfolds the tree.

In programs which have relatively balanced trees (e.g. divide and conquer computations), subtrees closer to the root are typically larger; work stealing exploits this property by stealing the oldest task in the victim's pool. Of the tasks in the pool, this one is always the one closest to the root of the computation. Much of the appeal of work stealing comes from its ability to steal a single task[1] representing a large computation.

When a thief is about to steal work, different potential victims can provide different amounts of work depending on the size of the subtree rooted at the oldest task in their respective task pools. Typically, thieves select victims at random. This is not necessarily optimal. The bigger the stolen sub tree, the fewer steals will be needed for the entire computation, minimizing overhead. In fact, if there are more workers with small amounts of work, a thief is likely to pick a victim with a less than average sized oldest task, which will make work even more unevenly distributed.

One possible response to this problem is sampling several workers before stealing and choosing the one with the largest available sub tree. This is similar in spirit to other load balancing algorithms that use sampling [93], although we do it to find a large pool to extract work from rather than finding a small queue for inserting work into.

[1]In a cache coherent system, executing the stolen task may yield extra cache misses for transferring the working set of the task to the thief's cache, but for many programs, the amount of extra misses grows more slowly than the task size due to data reuse within tasks.

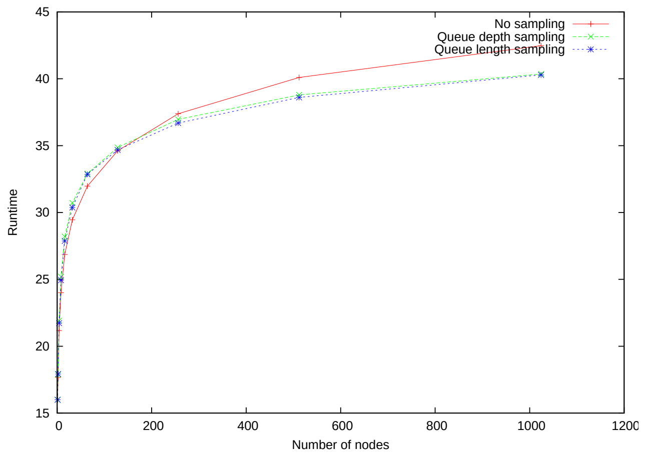


Figure 11.1: Simulated effects of sampling on execution time

While the run times of the tasks are not known in advance, a possible heuristic for balanced computations is their distance from the root of the tree. This simply extends the heuristic of stealing the oldest task of the given victim to the selection of the victim itself. Another heuristic that has been proposed is the number of tasks in the pool [28].

We have implemented a sampling victim selector for Wool, finding that the overhead of sampling (about half of the overhead of a steal) is significant enough that sampling should be disabled in situations with low stealing success rate. That is, when most workers have no tasks to offer, many samples will contain no useful information. Instead, it is better to steal the first task one finds. We implement this strategy by aborting sampling when an empty task pool is sampled.

Simulating sampling

In order to gain insight about the impact of sampling on the overall performance, both in terms of runtime and number of stolen tasks, we construct a simulation model of the system. The model consists of N parallell cores/nodes who either executes tasks if the local task queue is non empty and otherwise tries to steal work with a rate λ_{steal}. We assume that the points in time when different nodes attempt to steal work are independent which enables us to model the steal process as a Poisson process with parameter λ_{steal}. We model the sample process in a similar way with average rate $\lambda_{sample} = 2\lambda_{steal}$ (taking a sample is less costly than stealing since it only involves reads). The benchmark we study in this simulation setup is `tree` (given in figure 11.2), which builds a balanced binary tree of tasks. We use a tree depth 2 greater than the logarithm of the number of available nodes (each node will, on average, execute four leaves). hence larger number of nodes execute

```
VOID_TASK_2( tree, int, n, int, d )
{
  if( d>0 ) {
    SPAWN( tree, n, d-1 );
    CALL( tree, n, d-1);
    SYNC( tree );
  } else {
    loop( n );
  }
}
```

`loop(n)` computes for 2`n` cycles, making no memory references.

Figure 11.2: The `tree` task of the `stress` program

proportinally more work. Comparing again with measurements we conclude that the time to complete a leaf task is about the same order of magnitude as the time it takes to complete a stealing cycle letting us set $\lambda_{work} = \lambda_{steal}$. In the setup each node samples x independent nodes in order to determine which node to steal from, x being the *sample size*.

Figure 11.1 shows the runtime. First we note that benefit from using sampling in system with moderate number of cores is limited. However, as the system size grows the relative speedup from using sampling does too. We have found that 2 is the optimal sample size when $\lambda_{sample} = 2\lambda_{steal}$. Even if more samples results in a better victim it does not compensate for the overhead of taking the extra samples.

11.3 Work stealing

The stealing protocol

The TILEPro64 does not support the compare and swap primitive assumed by many work stealing algorithms [8, 57, 23], including the original direct task stack [44]. To bridge the gap, we modified the direct task stack by splitting the function of the `state` field into two fields, `state` and `alarm`. Figure 11.3 gives the main operations of the Two Field Direct Task Stack.

The main data structure of both versions of the direct task stack is an array of fixed size task descriptors. In contrast to most task stealers, synchronization among thieves and between thief and victim does not use the top and bottom pointers into the stack (indeed, the top pointer is private to the owner of the stack). Instead, a *state* field (in the new version also an `alarm` field) in each task descriptor is used. This means that the `bot` pointers are accessed without explicit synchronization. Instead, the `bot` pointer is logically owned by the worker that owns the task it

```
spawn_f( T_1 a_1, ..., T_n a_n ) {
  top->a_1 = a_1;
  ...
  top->a_n = a_n;
  top->alarm = NOT_STOLEN;
  store_fence();
  top->state = TASK( wrap_f );
  top++;
}

join_f( ) {
  State s;
  top--;
  s = test_and_set_to_EMPTY( &(top->state) );
  if( s != EMPTY )
    return f( top->a_1, ..., top->a_n );
  else {
    RTS_join( top );
    return top->result;
  }
}

RTS_join( Task *t ) {
  State s = t->state;
  Alarm a = t->alarm;
  do {
    while( s == EMPTY && a == NOT_STOLEN ) {
      s = t->state;
      a = t->alarm;
    }
    if( s != EMPTY )
      s = test_and_set_to_EMPTY( &(t->state) );
  } while( s == EMPTY && a == NOT_STOLEN );
  if( s != EMPTY )
    get_wrapper(s)(t);
  else if( a != DONE )
    while( t->alarm != DONE )
      steal( get_thief(s) );
  bot--;
}

RTS_steal( Worker *victim ) {
  Task *t = victim->bot;
  State s = t->state;
  if( is_task(s) ) {
    s = test_and_set_to_EMPTY( &(t->state) );
    if( s == EMPTY || victim->bot != t) {
       if( s != EMPTY ) t->state = s;
    } else {
      t->alarm = STOLEN( self_idx );
      victim->bot = t+1;
      get_wrapper(s)(t);
      store_fence();
      t->alarm = DONE;
    }
  }
}
```

1

Figure 11.3: The Two Field algorithm

points to. This can lead to the use of stale `bot` pointers which requires that the algorithm can back out of a steal that has used a stale `bot` pointer [44].

Logically, a task descriptor can be in one of the following states (these are the same as in the original direct task stack):

Empty There is no task stored in the descriptor and it is free to be reused.

Task There is a task in the descriptor that has not been stolen or aquired by the owner.

Busy This is a transient state where the task in the task descriptor has been acquired by a thief, but the thief has not yet commited to stealing it by writing its index into `state` (or `alarm` in the new version). This is represented with the same value of the `s`tate field as the EMPTY state.

Stolen The task is stolen but not yet completed.

Done The task was stolen but the thief has completed execution.

In the two field algorithm, these states are represented using a combination of the `s`tate and `a`larm fields, with the `s`tate field used for mutual exclusion when stealing and `a`larm used for communication between a successful thief and its victim.

The `spawn_f()` and `join_f()` functions are *task specific* and generated from the task definition by the `TASK_n` macros. They are similar to the corresponding functions in the original direct task stack algorithm. The join_f() function atomically checks that the task is not stolen, setting its state to EMPTY. If the task is stolen, it calls a function in the run-time system that handles synchronization with stolen tasks. This function spins until the thief either backs out of the steal or commits by writing its index into the `a`larm field of the task. This index is then used for *leap frogging* [115] until the stolen task is completed.

The RTS_steal() function reads the `b`ot pointer of the victim, then does a test and set on the indicated task. If the thief succeeds in acquiring the task, it re-reads `b`ot to see that the value it used was not stale, either committing to the steal or aborting it (that typically happens in less than 1% of successful steals).

Victim sets

One consequence of emphasizing the number of cores over the resources devoted to each core is that the amount of cache and TLB space in each core shrinks in relation to the number of potential stealing victims of that core. For a work stealer like Wool that polls aggressively for work, this may generate a large number of cache and TLB misses with an attendant increase in memory and inter core traffic, especially in situations with little available work and many failed steal attempts. In addition, the number of sharers of each cache line increases, increasing invalidation traffic.

To address this problem, we have designed the *victim set* algorithm for victim selection. When stealing is initiated (at the start of the computation and after each successful steal), the thief picks a random subset of the other workers and steals only from these. The size of the victim set should be large enough that, as long as there is at least one active thief in the system, every worker is reasonably often visited by a thief, but preferably not large enough to cause the problems discussed above. The optimal size thus depends on both machine parameters and the number of active thieves. We choose a set size as $\max(C, \frac{m \times p}{t})$ where C is the constant 12 for this machine (which has 16 data TLB entries), m is a multiplicity factor (how many active thieves we want per worker; we use 4 by default), p is the number of workers and t is the number of thieves. We have investigated the performance sensitivity of these parameters and found them to be relatively insensitive (similar values work just as well) as well as suitable for all of the benchmarks we have used.

The victim set is maintained as follows: Each worker has a private random permutation P of the indices of the other workers (which are its potential victims). When it initiates stealing, it picks a random starting point S in the permutation and attempts to steal from the workers it finds while proceeding circularly through the permutation. When it has encountered at least m thieves *and* made at least C steal attempts, it starts over from S for a possibly longer or shorter walk through P. If it wraps around to the position S again, it starts counting afresh. There is always a small probability that a set of thieves I becomes *isolated* from the rest by picking victim sets contained in I itself. For this reason, we reinitiate stealing, choosing a new random starting point S, after 1000 failed attempts.

11.4 Experimental results

All experiments in this paper were performed on a TileExpress board featuring a 700MHz TILEPro64 running Tilera Linux. We used a hypervisor configuration with 63 generally useable tiles and 1 tile dedicated to running the driver for the PCI interface over which the board communicates with the host.

For our experimental evaluation, we use relatively simple programs taken from the BOTS benchmark suite [41] as well as some of the programs used in earlier work on Wool [44]. The latter programs have small kernels that are repeated to get at least a second of parallel run time. We use them to study the effect of granularity by varying the size of the parallel computation and (inversely) the number of repetitions of the kernel. We thus get several workloads for each of these programs.

Table 1 gives some statistics of the programs. We see that the programs are unusually fine grained to be run on this number of cores. The **RepSz** column gives the run time in thousands of clock cycles for one repetition of the benchmark when executed on a single processor. For each such repetion, work is distributed over the machine followed in the end of joining together all of the work (much as in a tree barrier).

Params	Reps	Avg. Parallelism		RepSz	Granularities								
		0	1000		S_T	I_S^7	I_S^{15}	I_S^{23}	I_S^{31}	I_S^{39}	I_S^{47}	I_S^{55}	I_S^{63}
`cholesky`, parameters: number of rows													
125	1K	11.7	10.4	42841	7008	37	21	16	14	13	12	12	11
250	256	29.6	26.0	280848	6824	85	41	29	24	21	19	18	16
500	64	81.8	71.0	2073750	6732	233	101	67	52	44	38	34	31
1k	16	226.5	200.0	17013938	6950	761	307	196	148	120	102	89	80
2k	4	546.3	503.5	157477250	7132	2857	1121	688	510	405	338	292	257
`ssf`, parameters: number of concatenations													
12	16K	92.8	52.9	928	6489	31	17	13	11	10	10	9	9
13	8K	149.4	104.1	2661	11469	78	41	30	25	22	20	19	18
14	4K	245.3	192.1	7591	20190	177	87	64	53	47	42	39	36
15	2K	403.6	344.0	21557	35398	396	190	139	114	99	89	82	77
16	1K	645.8	584.8	60908	61773	1017	474	339	275	237	213	194	179
`stress`, leaf size 256 iterations, parameters: tree height													
10	64K	278.4	59.7	601	587	13	6	5	4	3	3	3	3
11	32K	404.8	102.7	1201	587	21	9	7	6	5	4	4	4
12	16K	534.3	170.6	2402	587	32	14	10	8	7	6	5	5
13	8K	631.3	269.5	4805	587	50	22	15	12	10	9	8	7
14	4K	1099.6	480.5	9610	587	80	33	23	18	15	13	12	11
`multisort`, parameters: array size													
16K	4K	68.3	31.0	2707	1142	16	8	6	4	4	3	3	3
64K	1K	108.8	73.8	11512	4645	58	27	19	15	13	12	11	10
256K	256	148.9	129.7	52500	18130	233	108	75	60	51	45	41	38
1M	64	139.9	137.4	279891	61626	1062	480	335	266	226	199	179	166
4M	16	91.5	91.7	1442000	122623	4510	1950	1343	1032	867	759	684	624
`fib`, parameters: n													
43	1	116802.5	110982.2	44121000	63	27042	1216	1045	1264	817	859	753	616
`uts`, parameters: sample tree													
T3	1	1087.6	682.4	8491000	2359	240	93	55	40	33	29	29	27
T3L	1	4457.1	2421.6	228676000	2567	316	79	46	35	35	32	29	31
`nqueens`, parameters: n													
13	1	65311.8	27140.4	31171000	521	13947	10399	6066	3876	3241	2600	2110	2072

Params are the parameters of each iteration, **Reps** is the number of repetitions, **Avg. Parallelism** is T_1/T_∞ assuming stealing overhead zero or 1000 cycles, **RepSz** is the size in 1000s of cycles of each repetition, **Granularities** are the average task size S_T in cycles and the average steal interval I_S in 1000s of cycles for 7 to 63 processors. Throughout, k is 1000 and K is 1024.

Table 1. Characteristics of the benchmarks

cholesky Sparse matrix factorization on a random square matrix using explicit nested tasks. Taken from the Cilk-5 distribution. Parameters are the number of matrix rows and the number of nonzero elements.

ssf Based on the Sub String Finder example from the TBB distribution. For each position in a string, it finds from which other position the longest identical substring starts. The string is given by the recursion $s_n = s_{n-1}s_{n-2}$ with $s_0 = "a"$ and $s_1 = "b"$ where n is the parameter in the workload.

stress A micro benchmark written to have a precisely controllable parallelism and granularity (the code is given in figure 11.2). The program creates a balanced binary tree of tasks with each leaf executing a simple loop making no memory references. A stress workload labelled `n d r` makes `r` sequential task trees

`CALL( tree, n, d )`. The granularity of the leaf tasks is $2 \times \texttt{n}$ cycles and the granularity of the parallel regions is $2 \times \texttt{n} \times 2^{\texttt{d}}$ cycles. We have `n` as 256 and we use `d` as scaling parameter.

fib Computes the Fibonnaci function using the naive algorithm, taken from the BOTS suite. We've included it as an example of a program with extremely fine grained tasks which exposes the behavior of inlined tasks. The input is 43.

uts The Unbalanced Tree Search program has been used to benchmark OpenMP task implementations [96] and as a challenge for a distributed memory work stealer [38]. It is also included in BOTS. The program builds a deep and unbalanced tree using the SHA-1 algorithm as a splittable random number generator. A node in the tree has either n or 0 children, the former with a probability very close to $\frac{1}{n}$. We use two workloads, T3 with 4112897 nodes and depth 1572 and T3L with 111345631 nodes and depth 17844., in both cases running with the smallest computational granularity (one iteration of the SHA-1 algorithm).

multisort The program is originally from the Cilk-5 distribution, and is also in BOTS. It sorts an array of integers using a combination of sorting algorithms. The outermost is a parallel merge sort which is replaced by a serial sort for sub arrays under a threshold size.

nqueens Also a BOTS program originating with Cilk-5. Solves the NQueens problem using a straight forward depth first search. N=13.

Looking at stress with a tree depth of 10, we see that the sequential run time of a repetition is just 601k cycles, or less than 10k cycles per core for the 63 core case. Looking at the rightmost column, we see that we have about 3k cycles of sequential execution between steals, making load balancing a very frequent operation.

Figure 11.4 shows relative speedups for the benchmarks. We choose relative speedup since our main focus is with the parallel scaling behavior, rather than with demonstrating the low overheads of Wool. For the scaling workloads we show increasing granularities in the plots from left to right. Each graph shows one victim selection strategy:

RAND A random number is generated for each steal attempt.

SAMP For p workers, up to about the square root of p workers are sampled. Sampling is not performed after failed steal attempts, and sampling is aborted (and a victim is selected among the already sampled workers) if a worker with no stealable work is sampled.

SET The victim set algorithm described in section 11.3.

SAMP+SET A combination of **SAMP** and **SET**. Sampling is abandoned when the set wraps around; that is, the same worker is never sampled twice in the same collection.

The grey area shows the expected speedup in a simple performance model based on the critical path length, or *span* of the computation [49].

Given a span t_∞ and a sequential execution time t_1 (not including stealing overhead), the low end of the speedup region for p processors in the figure is given by $\frac{t_1}{t_p}$ where t_p is the standard upper bound on the execution time of a work stealing computation:

$$t_p = t_\infty + \frac{t_1}{p}$$

The upper end is given by $\frac{t_1}{T_p}$ where T_p is the lower bound on execution time given as the maximum of t_∞ and $\frac{t_1}{p}$.

The difference between the upper and lower ends reflects the different shapes that a dependence graph with the same span and work can have. The upper end reflects a situation where the parallelism varies little over time; in particular, with no significant sequential periods, while the lower end reflects massive parallelism alternating with sequential execution. Our granularity scaling benchmarks, with their repeated parallel kernels, clearly belong in the latter category.

Overall, the programs scale close to the theoretical prediction which indicates that the Tilera processor has adequate resources and no obvious bottlenecks. It can support all the cores being active. There is a small tendency in several runs that the speedup falls away from the theoretical model at high core counts since the number of load balancing steals increases due to the imbalance in work distribution.

There are two programs that deviate significantly from the theoretical prediction: `uts` and `multisort`. A breakdown of their execution time shows that for `uts` the main problem is that the leap frogging strategy is not powerful enough to find work; a lot of time is spent in unsuccessful steal attempts in `RTS_join()`. For `sort` the issue is an increase in application time due to increased cache misses. This is a known problem; the program has little computaion per byte of data.

11.5 Conclusions and future work

We have studied the performance of the Wool work stealing task scheduler on a 64 core Tilera processor. We found that most programs scale well, being primarily limited by available parallelism. We have demonstrated that it is possible to achieve stealing oveheads counted in hundreds of cycles rather than thousands.

We have also studied the unbalancing effects of random work stealing and applied sampling to mitigate the problem. For a stealer with the low overheads of Wool, where sampling is only about twice as expensive as stealing, the reduction in the number of steals does not pay for the sampling overhead for 63 cores. Simula-

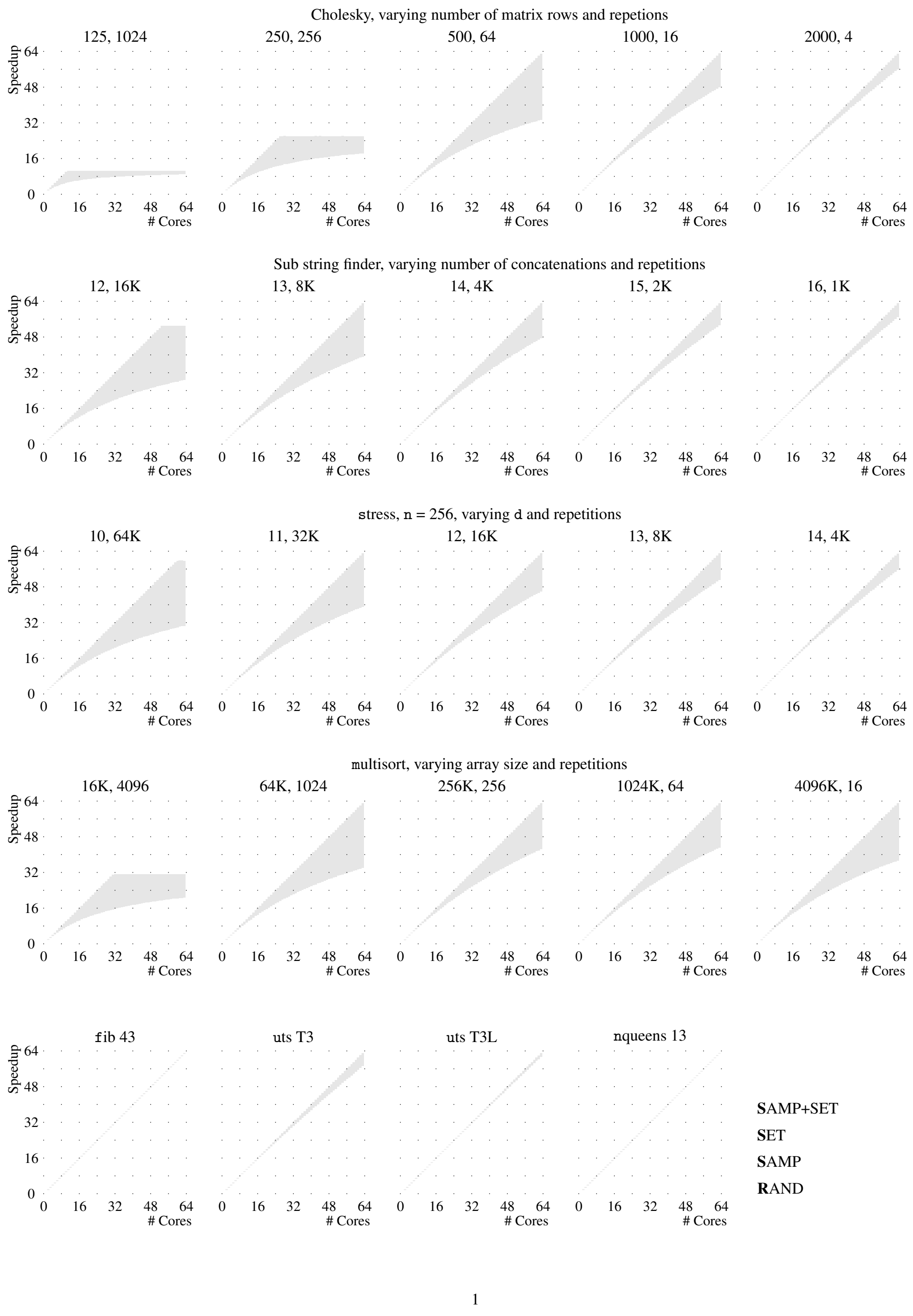


Figure 11.4: Speedup on a TILEPro64

tion results indicate that the imbalance grows with system size and that sampling will be more attractive when the core counts pass a hundred.

We have also presented two algorithms which adapt Wool to the Tilera processor; a task stack management algorithm which only needs a test-and-set primitive and a victim selection algorithm that improves cache and TLB locality, something that is important for a processor with many rather simple cores.

For the future we intend to investigate the use of polling and other techniqes for mitigating stealing imbalance, especially with a view to the trade-off between the information obtained by polling and the cost of obtaining it.

We will also investigate alternatives to leap frogging that do not require code generator support, but still are robust against the problems encountered by `uts`.

Bibliography

[1] www.bittorrent.com, The BitTorrent protocol.

[2] en.wikipedia.org/wiki/EDonkey2000, The eDonkey2000 file sharing application.

[3] *OpenMP application programming interface, version 3.0*, May 2008, Available from www.openmp.org.

[4] *The international conference on peer-to-peer computing. http://p2p-conference.org/*, 2013.

[5] Karl Aberer, Anwitaman Datta, and Manfred Hauswirth, *Efficient, self-contained handling of identity in peer-to-peer systems*, IEEE Trans. Knowl. Data Eng. **16** (2004), no. 7, 858–869.

[6] Abdelnaser Adas, *Traffic models in broadband networks*, Communications Magazine, IEEE **35** (1997), no. 7, 82–89.

[7] Bengt Ahlgren, Christian Dannewitz, Claudio Imbrenda, Dirk Kutscher, and Börje Ohlman, *A survey of information-centric networking*, Communications Magazine, IEEE **50** (2012), no. 7, 26–36.

[8] Nimar S. Arora, Robert D. Blumofe, and C. Greg Plaxton, *Thread scheduling for multiprogrammed multiprocessors*, SPAA '98: Proceedings of the tenth annual ACM symposium on Parallel algorithms and architectures (New York, NY, USA), ACM, 1998, pp. 119–129.

[9] Søren Asmussen, Soren Asmussen, and Sren Asmussen, *Applied probability and queues*, vol. 2, Springer New York, 2003.

[10] James Aspnes, Zoë Diamadi, and Gauri Shah, *Fault-tolerant routing in peer-to-peer systems*, Proceedings of the twenty-first annual symposium on Principles of distributed computing, ACM Press, 2002, pp. 223–232.

[11] Rana Bakhshi and Dilian Gurov, *Verification of peer-to-peer algorithms: A case study*, Electronic Notes in Theoretical Computer Science **181** (2007), 35–47.

[12] Anirban Basu, Simon Fleming, James Stanier, Stephen Naicken, Ian Wakeman, and Vijay K Gurbani, *The state of peer-to-peer network simulators*, ACM Computing Surveys (CSUR) **45** (2013), no. 4, 46.

[13] R. Bellman, *On a routing problem*, Quarterly of Applied Mathematics (1958), 16(1):87–90.

[14] Richard Bellman, *On a routing problem*, Tech. report, DTIC Document, 1956.

[15] C. Blake and R. Rodrigues, *High availability, scalable storage, dynamic peer networks: Pick two*, Proceedings of the 9th Workshop on Hot Topics in Operating Systems (HotOS'03), USENIX, 2003, pp. 1–6.

[16] Robert D Blumofe and Charles E Leiserson, *Scheduling multithreaded computations by work stealing*, Journal of the ACM (JACM) **46** (1999), no. 5, 720–748.

[17] Robert D. Blumofe and Charles E. Leiserson, *Scheduling multithreaded computations by work stealing*, J. ACM **46** (1999), no. 5, 720–748.

[18] Andrea Bobbio, Marco Gribaudo, and Miklós Telek, *Analysis of large scale interacting systems by mean field method*, Quantitative Evaluation of Systems, 2008. QEST'08. Fifth International Conference on, IEEE, 2008, pp. 215–224.

[19] Gunter Bolch, Stefan Greiner, Hermann de Meer, and Kishor Shridharbhai Trivedi, *Queueing networks and markov chains: modeling and performance evaluation with computer science applications*, John Wiley & Sons, 2006.

[20] Jean-Yves Le Boudec, *The stationary behaviour of fluid limits of reversible processes is concentrated on stationary points*, arXiv preprint arXiv:1009.5021 (2010).

[21] Stephen Boyd, Arpita Ghosh, Balaji Prabhakar, and Devavrat Shah, *Randomized gossip algorithms*, Information Theory, IEEE Transactions on **52** (2006), no. 6, 2508–2530.

[22] L. Breslau, P. Cao, L. Fan, G. Phillips, and S. Shenker, *Web caching and zipf-like distributions: Evidence and implications*, INFOCOM'99. Eighteenth Annual Joint Conference of the IEEE Computer and Communications Societies. Proceedings. IEEE, vol. 1, IEEE, 1999, pp. 126–134.

[23] David Chase and Yossi Lev, *Dynamic circular work-stealing deque*, SPAA '05: Proceedings of the seventeenth annual ACM symposium on Parallelism in algorithms and architectures (New York, NY, USA), ACM, 2005, pp. 21–28.

[24] H. Che, Y. Tung, and Z. Wang, *Hierarchical web caching systems: Modeling, design and experimental results*, Selected Areas in Communications, IEEE Journal on **20** (2002), no. 7, 1305–1314.

[25] Shimin Chen, Phillip B. Gibbons, Michael Kozuch, Vasileios Liaskovitis, Anastassia Ailamaki, Guy E. Blelloch, Babak Falsafi, Limor Fix, Nikos Hardavellas, Todd C. Mowry, and Chris Wilkerson, *Scheduling threads for constructive cache sharing on cmps*, SPAA '07: Proceedings of the nineteenth annual ACM symposium on Parallel algorithms and architectures (New York, NY, USA), ACM, 2007, pp. 105–115.

[26] *Cisco visual networking index: Forecast and methodology, 2010-2015*, June 2011.

[27] Florence Clévenot and Philippe Nain, *A simple fluid model for the analysis of the squirrel peer-to-peer caching system*, INFOCOM 2004. Twenty-third AnnualJoint Conference of the IEEE Computer and Communications Societies, vol. 1, IEEE, 2004.

[28] Gilberto Contreras and Margaret Martonosi, *Characterizing and improving the performance of the intel threading building blocks runtime system*, International Symposium on Workload Characterization (IISWC 2008), September 2008.

[29] Robert M Corless, Gaston H Gonnet, David EG Hare, David J Jeffrey, and Donald E Knuth, *On the lambertw function*, Advances in Computational mathematics **5** (1996), no. 1, 329–359.

[30] S.A. Crosby and D.S. Wallach, *An analysis of bittorrents two kademlia-based dhts*, Tech. report, Technical Report TR07-04, Rice University, 2007.

[31] F. Dabek, J. Li, E. Sit, J. Robertson, M.F. Kaashoek, and R. Morris, *Designing a dht for low latency and high throughput*, Proc. NSDI **4** (2004).

[32] L. D'Acunto, J.A. Pouwelse, and H.J. Sips, *A measurement of NAT and firewall characteristics in peer-to-peer systems*, Proc. 15-th ASCI Conference (P.O. Box 5031, 2600 GA Delft, The Netherlands) (Lex Wolters Theo Gevers, Herbert Bos, ed.), Advanced School for Computing and Imaging (ASCI), June 2009, pp. 1–5.

[33] Cameron Dale and Jiangchuan Liu, *apt-p2p: A peer-to-peer distribution system for software package releases and updates*, IEEE INFOCOM (Rio de Janeiro, Brazil), April 2009.

[34] Mads. Dam and Rolf. Stadler, *A generic protocol for network state aggregation*, In Proc. Radiovetenskap och Kommunikation (RVK) (2005), 14–16.

[35] A. Dan and D. Towsley, *An approximate analysis of the lru and fifo buffer replacement schemes*, ACM SIGMETRICS Performance Evaluation Review **18** (1990), no. 1, 143–152.

[36] G. Dán and N. Carlsson, *Power-law revisited: large scale measurement study of p2p content popularity*, Proceedings of the 9th international conference on Peer-to-peer systems, USENIX Association, 2010, pp. 12–12.

[37] Michael J. Demmer and Maurice Herlihy, *The arrow distributed directory protocol*, DISC (Shay Kutten, ed.), Lecture Notes in Computer Science, vol. 1499, Springer, 1998, pp. 119–133.

[38] James Dinan, Sriram Krishnamoorthy, D. Brian Larkins, Jarek Nieplocha, and P. Sadayappan, *Scalable work stealing*, SC09, ACM, November 2009.

[39] S. Dolev, *Self-stabilization*, The MIT press, 2000.

[40] Shlomi Dolev, Amos Israeli, and Shlomo Moran, *Self-stabilization of dynamic systems assuming only read/write atomicity*, Distributed Computing **7** (1993), no. 1, 3–16.

[41] Alejandro Duran, Xavier Teruel, Roger Ferrer, Xavier Martorell, and Eduard Ayguadé, *Barcelona openmp tasks suite: A set of benchmarks targeting the exploitation of task parallelism in openmp*, ICPP, 2009, pp. 124–131.

[42] Cynthia Dwork, Nancy Lynch, and Larry Stockmeyer, *Consensus in the presence of partial synchrony*, Journal of the ACM (JACM) **35** (1988), no. 2, 288–323.

[43] Agner Krarup Erlang, *The theory of probabilities and telephone conversations*, Nyt Tidsskrift for Matematik B **20** (1909), no. 33-39, 16.

[44] Karl-Filip Faxén, *Efficient work stealing for fine grained parallelism*, Proc. of 39th International Conference on Parallel Processing (San Diego), 2010.

[45] W. Feller, *An Introduction to Probability Theory and its Applications, Vol 1*, Wiley, Singapore, 1968.

[46] Michael J. Freedman, Karthik Lakshminarayanan, Sean Rhea, and Ion Stoica, *Non-transitive connectivity and DHTs*, WORLDS'05: Proceedings of the 2nd conference on Real, Large Distributed Systems (Berkeley, CA, USA), USENIX Association, 2005, pp. 55–60.

[47] C. Fricker, P. Robert, and J. Roberts, *A versatile and accurate approximation for lru cache performance*, Arxiv preprint arXiv:1202.3974 (2012).

[48] C. Fricker, P. Robert, J. Roberts, and N. Sbihi, *Impact of traffic mix on caching performance in a content-centric network*, Arxiv preprint arXiv:1202.0108 (2012).

[49] Matteo Frigo, Charles E. Leiserson, and Keith H. Randall, *The implementation of the Cilk-5 multithreaded language*, SIGPLAN Conference on Programming Language Design and Implementation, 1998, pp. 212–223.

[50] P.B. Godfrey, S. Shenker, and I. Stoica, *Minimizing churn in distributed systems*, Proceedings of the 2006 conference on Applications, technologies, architectures, and protocols for computer communications (2006), 147–158.

[51] R. Guerraoui and L. Rodrigues, *Reliable distributed programming*, Springer Verlag, Berlin, 2006.

[52] Rachid Guerraoui and Maysam Yabandeh, *Model checking a networked system without the network*, Proceedings of the 8th USENIX conference on Networked Systems Design and Implementation, NSDI, vol. 11, 2011.

[53] K. Gummadi, R. Gummadi, S. Gribble, S. Ratnasamy, S. Shenker, and I. Stoica, *The impact of dht routing geometry on resilience and proximity*, Proceedings of the 2003 conference on Applications, technologies, architectures, and protocols for computer communications (2003), 381–394.

[54] K.P. Gummadi, S. Saroiu, and S.D. Gribble, *King: estimating latency between arbitrary internet end hosts*, Proceedings of the second ACM SIGCOMM Workshop on Internet measurment (2002), 5–18.

[55] Krishna P. Gummadi, Richard J. Dunn, Stefan Saroiu, Steven D. Gribble, Henry M. Levy, and John Zahorjan, *Measurement, modeling, and analysis of a peer-to-peer file-sharing workload*, SIGOPS Oper. Syst. Rev. **37** (2003), no. 5, 314–329.

[56] G. Haßlinger and F. Hartleb, *Content delivery and caching from a network provider's perspective*, Computer Networks **55** (2011), no. 18, 3991–4006.

[57] Danny Hendler and Nir Shavit, *Non-blocking steal-half work queues*, PODC '02: Proceedings of the twenty-first annual symposium on Principles of distributed computing (New York, NY, USA), ACM, 2002, pp. 280–289.

[58] H. Peter Hofstee, *Power efficient processor architecture and the cell processor*, High-Performance Computer Architecture, International Symposium on **0** (2005), 258–262.

[59] Yan Huang, Tom Z.J. Fu, Dah-Ming Chiu, John C.S. Lui, and Cheng Huang, *Challenges, design and analysis of a large-scale p2p-vod system*, SIGCOMM Comput. Commun. Rev. **38** (2008), no. 4, 375–388.

[60] R. Jimenez, F. Osmani, and B. Knutsson, *Connectivity properties of mainline BitTorrent DHT nodes*, Peer-to-Peer Computing, 2009. P2P '09. IEEE Ninth International Conference on, 2009, pp. 262–270.

[61] J. A. Kahle, M. N. Day, H. P. Hofstee, C. R. Johns, T. R. Maeurer, and D. Shippy, *Introduction to the cell multiprocessor*, IBM J. Res. Dev. **49** (2005), 589–604.

[62] R. Karedla, J.S. Love, and B.G. Wherry, *Caching strategies to improve disk system performance*, Computer **27** (1994), no. 3, 38–46.

[63] F. P. Kelly, *Reversibility and stochastic networks*, John Wiley & Sons New York, 1979.

[64] David G Kendall, *Stochastic processes occurring in the theory of queues and their analysis by the method of the imbedded markov chain*, The Annals of Mathematical Statistics (1953), 338–354.

[65] Anne-Marie Kermarrec, Alessio Pace, Vivien Quema, and Valerio Schiavoni, *NAT-resilient gossip peer sampling*, Distributed Computing Systems, International Conference on **0** (2009), 360–367.

[66] Gerwin Klein, Kevin Elphinstone, Gernot Heiser, June Andronick, David Cock, Philip Derrin, Dhammika Elkaduwe, Kai Engelhardt, Rafal Kolanski, Michael Norrish, et al., *sel4: Formal verification of an os kernel*, Proceedings of the ACM SIGOPS 22nd symposium on Operating systems principles, ACM, 2009, pp. 207–220.

[67] Leonard Kleinrock, *Queueing systems: I theory*, Wiley-interscience, 1975.

[68] ———, *Queueing systems: Ii computer applications*, John Wiley and Sons New York, 1975.

[69] Steven Y. Ko, Imranul Hoque, and Indranil Gupta, *Using tractable and realistic churn models to analyze quiescence behavior of distributed protocols*, SRDS '08: Proceedings of the 2008 Symposium on Reliable Distributed Systems (Washington, DC, USA), IEEE Computer Society, 2008, pp. 259–268.

[70] Joseph S. Kong, Jesse S. A. Bridgewater, and Vwani P. Roychowdhury, *A general framework for scalability and performance analysis of dht routing systems*, DSN '06: Proceedings of the International Conference on Dependable Systems and Networks (Washington, DC, USA), IEEE Computer Society, 2006, pp. 343–354.

[71] S. Krishnamurthy, S. El-Ansary, E. Aurell, and S. Haridi, *A statistical theory of chord under churn*, Peer-to-Peer Systems IV (2005), 93–103.

[72] Supriya Krishnamurthy, Sameh El-Ansary, Erik Aurell, and Seif Haridi, *Comparing maintenance strategies for overlays*, Tech. report, Swedish Institute of Computer Science, Proceeedings of PDP2008 2007.

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

[74] Fabian Kuhn and Roger Wattenhofer, *Dynamic analysis of the arrow distributed protocol*, SPAA (Phillip B. Gibbons and Micah Adler, eds.), ACM, 2004, pp. 294–301.

[75] Rakesh Kumar, Yong Liu, and Keith W. Ross, *Stochastic fluid theory for p2p streaming systems*, INFOCOM, 2007, pp. 919–927.

[76] G. Latouche and PG Taylor, *A stochastic fluid model for an ad hoc mobile network*, Queueing Systems **63** (2009), no. 1, 109–129.

[77] Daan Leijen, Wolfram Schulte, and Sebastian Burckhardt, *The design of a task parallel library*, SIGPLAN Not. **44** (2009), no. 10, 227–242.

[78] Charles E. Leiserson, *The cilk++ concurrency platform*, DAC '09: Proceedings of the 46th Annual Design Automation Conference (New York, NY, USA), ACM, 2009, pp. 522–527.

[79] Derek Leonard, Vivek Rai, and Dmitri Loguinov, *On lifetime-based node failure and stochastic resilience of decentralized peer-to-peer networks*, SIGMETRICS Perform. Eval. Rev. **33** (2005), no. 1, 26–37.

[80] B. Li, Y. Qu, Y. Keung, S. Xie, C. Lin, J. Liu, and X. Zhang, *Inside the New Coolstreaming: Principles, Measurements and Performance Implications*, INFOCOM 2008. The 27th Conference on Computer Communications. IEEE, 2008.

[81] Jinyang Li, Jeremy Stribling, Robert Morris, M. Frans Kaashoek, and Thomer M. Gil, *A performance vs. cost framework for evaluating dht design tradeoffs under churn*, Proceedings of the 24th Infocom (Miami, FL), March 2005.

[82] Taoyu Li, Minghua Chen, Dah-Ming Chiu, and Maoke Chen, *Queuing models for peer-to-peer systems.*, IPTPS, 2009, p. 4.

[83] D. Liben-Nowell, H. Balakrishnan, and D. Karger, *Analysis of the evolution of peer-to-peer systems*, Proceedings of the twenty-first annual symposium on Principles of distributed computing, ACM, 2002, pp. 233–242.

[84] Yangyang Liu and Jianping Pan, *The impact of NAT on BitTorrent-like p2p systems*, Peer-to-Peer Computing, 2009. P2P '09. IEEE Ninth International Conference on, 2009, pp. 242–251.

[85] Eng Keong Lua, Jon Crowcroft, Marcelo Pias, Ravi Sharma, and Steven Lim, *A survey and comparison of peer-to-peer overlay network schemes.*, IEEE Communications Surveys and Tutorials **7** (2005), no. 1-4, 72–93.

[86] Samuel Madden, Michael J. Franklin, Joseph M. Hellerstein, and Wei Hong, *Tag: A tiny aggregation service for ad-hoc sensor networks*, OSDI, 2002.

[87] Petar Maymounkov and David Mazieres, *Kademlia: A peer-to-peer information system based on the xor metric*, The 1st Interational Workshop on Peer-to-Peer Systems (IPTPS'02), 2002, http://www.cs.rice.edu/Conferences/IPTPS02/.

[88] Boris Mejías and Peter Van Roy, *The relaxed-ring: a fault-tolerant topology for structured overlay networks*, Parallel Processing Letters **18** (2008), no. 3, 411–432.

[89] Sean P Meyn, *Control techniques for complex networks*, Cambridge University Press, 2008.

[90] Sean P Meyn and Richard L Tweedie, *Stability of markovian processes iii: Foster-lyapunov criteria for continuous-time processes*, Advances in Applied Probability (1993), 518–548.

[91] P. Van Mieghem, *Performance Analysis of Communication Networks and Systems*, Cambridge Universty Press, New York, 2005.

[92] Michael Mitzenmacher and Eli Upfal, *Probability and computing: Randomized algorithms and probabilistic analysis*, Cambridge University Press, 2005.

[93] Michael David Mitzenmacher, *The power of two choices in randomized load balancing*, IEEE Transactions on Parallel and Distributed Systems **12** (2001), no. 10.

[94] R. Nelson, *The mathematics of product form queueing networks*, ACM Computing Survey **25** (1993), no. 3, 339–369.

[95] Bernt Øksendal, *Stochastic differential equations*, Springer, 2003.

[96] Stephen L. Olivier and Jan F. Prins, *Evaluating OpenMP 3.0 run time systems on unbalanced task graphs*, IWOMP '09: Proceedings of the 5th International Workshop on OpenMP (Berlin, Heidelberg), Springer-Verlag, 2009, pp. 63–78.

[97] David Peleg and Eilon Reshef, *A variant of the arrow distributed directory with low average complexity*, ICALP (Jirí Wiedermann, Peter van Emde Boas, and Mogens Nielsen, eds.), Lecture Notes in Computer Science, vol. 1644, Springer, 1999, pp. 615–624.

[98] Dongyu Qiu and Rayadurgam Srikant, *Modeling and performance analysis of bittorrent-like peer-to-peer networks*, SIGCOMM, 2004, pp. 367–378.

[99] James Reinders, *Intel threading building blocks*, O'Reilly & Associates, Inc., Sebastopol, CA, USA, 2007.

[100] J. Reineke and D. Grund, *Relative competitiveness of cache replacement policies*, ACM SIGMETRICS Performance Evaluation Review **36** (2008), no. 1, 431–432.

[101] S. Rhea, B.G. Chun, J. Kubiatowicz, and S. Shenker, *Fixing the embarrassing slowness of opendht on planetlab*, Proc. WORLDS (2005).

[102] S. Rhea, B. Godfrey, B. Karp, J. Kubiatowicz, S. Ratnasamy, S. Shenker, I. Stoica, and H. Yu, *OpenDHT: A public DHT service and its uses*, 2005.

[103] Sean Rhea, Dennis Geels, Timothy Roscoe, and John Kubiatowicz, *Handling churn in a DHT*, Proceedings of the 2004 USENIX Annual Technical Conference(USENIX '04) (Boston, Massachusetts, USA), June 2004.

[104] Roberto Roverso, Sameh El-Ansary, and Seif Haridi, *NATCracker: NAT combinations matter*, Computer Communications and Networks, International Conference on **0** (2009), 1–7.

[105] Larry Seiler, Doug Carmean, Eric Sprangle, Tom Forsyth, Michael Abrash, Pradeep Dubey, Stephen Junkins, Adam Lake, Jeremy Sugerman, Robert Cavin, Roger Espasa, Ed Grochowski, Toni Juan, and Pat Hanrahan, *Larrabee: a many-core x86 architecture for visual computing*, ACM Trans. Graph. **27** (2008), no. 3, 1–15.

[106] Tallat M. Shafaat, Monika Moser, Thorsten Schütt, Alexander Reinefeld, Ali Ghodsi, and Seif Haridi, *Key-based consistency and availability in structured overlay networks*, Proceedings of the 3rd International ICST Conference on Scalable Information Systems (Infoscale'08), ACM, jun 2008.

[107] Devavrat Shah, *Gossip algorithms*, Now Publishers Inc, 2009.

[108] Bittorrent Protocol Specification, 2013, http://wiki.theory.org/BitTorrentSpecification.

[109] Ion Stoica, Robert Morris, David Karger, M Frans Kaashoek, and Hari Balakrishnan, *Chord: A scalable peer-to-peer lookup service for internet applications*, ACM SIGCOMM Computer Communication Review, vol. 31, ACM, 2001, pp. 149–160.

[110] Daniel Stutzbach and Reza Rejaie, *Understanding churn in peer-to-peer networks*, IMC '06: Proceedings of the 6th ACM SIGCOMM conference on Internet measurement (New York, NY, USA), ACM, 2006, pp. 189–202.

[111] Jim Sukha, *Brief announcement: a lower bound for depth-restricted work stealing*, SPAA '09: Proceedings of the twenty-first annual symposium on Parallelism in algorithms and architectures (New York, NY, USA), ACM, 2009, pp. 124–126.

[112] N. van Dijk, *Queueing Networks and Product Forms*, John Wiley, New York, 1993.

[113] N.G. van Kampen, *Stochastic processes in physics and chemistry*, North-Holland Publishing Company, 1981, ISBN-0-444-86200-5.

[114] Nicolaas Godfried Van Kampen, *Stochastic processes in physics and chemistry*, vol. 1, Access Online via Elsevier, 1992.

[115] David B. Wagner and Bradley G. Calder, *Leapfrogging: a portable technique for implementing efficient futures*, SIGPLAN Not. **28** (1993), no. 7, 208–217.

[116] Ferdinand Wagner, Ruedi Schmuki, Thomas Wagner, and Peter Wolstenholme, *Modeling software with finite state machines: a practical approach*, CRC Press, 2006.

[117] J. Wang, *A survey of web caching schemes for the internet*, ACM SIGCOMM Computer Communication Review **29** (1999), no. 5, 36–46.

[118] S. Wang, D. Xuan, and W. Zhao, *Analyzing and enhancing the resilience of structured peer-to-peer systems*, Journal of Parallel and Distributed Computing **65** (2005), no. 2, 207–219.

[119] J. Warland, *An Introduction to Queueing Networks*, Prentice-Hall, 1988.

[120] D.A.B. Weikle, S.A. McKee, and W.A. Wulf, *Caches as filters: A new approach to cache analysis*, Modeling, Analysis and Simulation of Computer and Telecommunication Systems, 1998. Proceedings. Sixth International Symposium on, IEEE, 1998, pp. 2–12.

[121] Jim Woodcock, Peter Gorm Larsen, Juan Bicarregui, and John Fitzgerald, *Formal methods: Practice and experience*, ACM Computing Surveys (CSUR) **41** (2009), no. 4, 19.

[122] Di Wu, Ye Tian, and Kam-Wing Ng, *An analytical study on optimizing the lookup performance of distributed hash table systems under churn*, Concurrency and Computation: Practice and Experience **19** (2007), no. 4, 543–569.

[123] Zhongmei Yao, Derek Leonard, Xiaoming Wang, and Dmitri Loguinov, *Modeling heterogeneous user churn and local resilience of unstructured p2p networks*, Network Protocols, 2006. ICNP'06. Proceedings of the 2006 14th IEEE International Conference on, IEEE, 2006, pp. 32–41.

[124] Pamela Zave, *Using lightweight modeling to understand chord*, ACM SIGCOMM Computer Communication Review **42** (2012), no. 2, 49–57.

Swedish Institute of Computer Science

SICS Dissertation Series

14. Sverker Jansson, AKL - - A Multiparadigm Programming Language, 1994.
15. Fredrik Orava, On the Formal Analysis of Telecommunication Protocols, 1994.
16. Torbjörn Keisu, Tree Constraints, 1994.
17. Olof Hagsand, Computer and Communication Support for Interactive Distributed Applications, 1995.
18. Björn Carlsson, Compiling and Executing Finite Domain Constraints, 1995.
19. Per Kreuger, Computational Issues in Calculi of Partial Inductive Definitions, 1995.
20. Annika Waern, Recognising Human Plans: Issues for Plan Recognition in Human-Computer Interaction, 1996.
21. Björn Gambäck, Processing Swedish Sentences: A Unification-Based Grammar and Some Applications, 1997.
22. Klas Orsvärn, Knowledge Modelling with Libraries of Task Decomposition Methods, 1996.
23. Kia Höök, A Glass Box Approach to Adaptive Hypermedia, 1996.
24. Bengt Ahlgren, Improving Computer Communication Performance by Reducing Memory Bandwidth Consumption, 1997.
25. Johan Montelius, Exploiting Fine-grain Parallelism in Concurrent Constraint Languages, 1997.
26. Jussi Karlgren, Stylistic experiments in information retrieval, 2000.
27. Ashley Saulsbury, Attacking Latency Bottlenecks in Distributed Shared Memory Systems, 1999.
28. Kristian Simsarian, Toward Human Robot Collaboration, 2000.
29. Lars-åke Fredlund, A Framework for Reasoning about Erlang Code, 2001.
30. Thiemo Voigt, Architectures for Service Differentiation in Overloaded Internet Servers, 2002.
31. Fredrik Espinoza, Individual Service Provisioning, 2003.
32. Lars Rasmusson, Network capacity sharing with QoS as a financial derivative pricing problem: algorithms and network design, 2002.
33. Martin Svensson, Defining, Designing and Evaluating Social Navigation, 2003.

34. Joe Armstrong, Making reliable distributed systems in the presence of software errors, 2003.

35. Emmanuel Frécon, DIVE on the Internet, 2004.

36. Rickard Cöster, Algorithms and Representations for Personalised Information Access, 2005.

37. Per Brand, The Design Philosophy of Distributed Programming Systems: the Mozart Experience, 2005.

38. Sameh El-Ansary, Designs and Analyses in Structured Peer-to-Peer Systems, 2005.

39. Erik Klintskog, Generic Distribution Support for Programming Systems, 2005.

40. Markus Bylund, A Design Rationale for Pervasive Computing - User Experience, Contextual Change, and Technical Requirements, 2005.

41. Åsa Rudström, Co-Construction of hybrid spaces, 2005.

42. Babak Sadighi Firozabadi, Decentralised Privilege Management for Access Control, 2005.

43. Marie Sjölinder, Age-related Cognitive Decline and Navigation in Electronic Environments, 2006.

44. Magnus Sahlgren, The Word-Space Model: Using Distributional Analysis to Represent Syntagmatic and Paradigmatic Relations between Words in High-dimensional Vector Spaces, 2006.

45. Ali Ghodsi, Distributed k-ary System: Algorithms for Distributed Hash Tables, 2006.

46. Stina Nylander, Design and Implementation of Multi-Device Services, 2007

47. Adam Dunkels, Programming Memory-Constrained Networked Embedded Systems, 2007

48. Jarmo Laaksolahti, Plot, Spectacle, and Experience: Contributions to the Design and Evaluation of Interactive Storytelling, 2008

49. Daniel Gillblad, On Practical Machine Learning and Data Analysis, 2008

50. Fredrik Olsson, Bootstrapping Named Entity Annotation by Means of Active Machine Learning: a Method for Creating Corpora, 2008

51. Ian Marsh, Quality Aspects of Internet Telephony, 2009

52. Markus Bohlin, A Study of Combinatorial Optimization Problems in Industrial Computer Systems, 2009

53. Petra Sundström, Designing Affective Loop Experiences, 2010

54. Anders Gunnar, Aspects of Proactive Traffic Engineering in IP Networks, 2011

55. Preben Hansen, Task-based Information Seeking and Retrieval in the Patent Domain: Process and Relationships, 2011

56. Fredrik Österlind, Improving low-power wireless protocols with timing-accurate simulation, 2011

57. Ahmad Al-Shishtawy, Self-Management for Large-Scale Distributed Systems, 2012

58. Henrik Abrahamsson, Network overload avoidance by traffic engineering and content caching, 2012

59. Mattias Rost, Mobility is the Message: Experiment with Mobile Media Sharing, 2013

60. Amir H. Payberah, Live Streaming in P2P and Hybrid P2P-Cloud Environments for the open Internet, 2013

61. Oscar Täckström, Predicting Linguistic Structure with Incomplete and Cross-Lingual Supervision, 2013

62. Cosmin Arad, Programming Model and Protocols for Reconfigurable Distributed Systems, 2013

63. Tallat M. Shafaat, Partition Tolerance and Data Consistency in Structured Overlay Networks, 2013

64. Shahid Raza, Lightweight Security Solutions for the Internet of Things, 2013

65. Mattias Jacobsson, Tinkering with Interactive Materials: Studies, Concepts and Prototypes, 2013

66. Baki Cakici, The Informed Gaze: On the Implications of ICT-Based Surveillance, 2013

67. John Ardelius, On the Performance Analysis of Large Scale, Dynamic, Distributed and Parallel Systems. 2013