A Massively Scalable Architecture For Instant Messaging & Presence

Jorrit Schippers

Master Telematics

Faculty of Electrical Engineering, Mathematics and Computer Science

University of Twente

Committee:

1. Dr. A.K.I. Remke (Design and Analysis of Communication Systems, University of Twente)
2. Dr. ir. M. Wegdam (Information Systems, University of Twente)
3. Prof. dr. ir. B.R.H.M. Haverkort (Design and Analysis of Communication Systems, University of Twente)
4. Drs. H. Punt (Hyves, Startphone Limited)

Day of the defense: July 10th, 2009

Abstract

In this thesis we design a scalable architecture for the instant messaging & presence service offered by Hyves. The current architecture consists of a variable number of application nodes and database slave nodes, which are located pair-wise on machines. Persistent presence information is distributed using two database master nodes. Network and database bottlenecks exist in the slave and master nodes, preventing the architecture to scale to future workloads.

We have developed a modelling and analysis approach to measure scalability properties of three new architectures. The first architecture was the result of applying database partitioning to the current architecture. The other two architectures are inspired by the architectures of Facebook chat and of Windows Live Messenger, which have been published by their respective creators. The Facebook-inspired architecture aggregates presence updates to reduce the amount of internal messages, whereas the Windows Live Messenger-inspired architecture uses subscriptions to coordinate presence propagation.

We use HIT to model the architectures and analyse the relation between workload and usage of databases and network links. It shows that the first architecture does not scale linearly: doubling the workload requires more than twice the number of machines. The second architecture scales sublinearly for increased workloads, but is influenced the most by changes in the total number of users or the portion of users online at peak times. The third architecture shows linear scalability, as it has a constant relation between workload and utilisation of resources.

We recommend the third, subscription-based alternative as a scalable replacement for the current architecture.

Acknowledgements

I would like to acknowledge the University of Dortmund and in particular Jürgen Mäter for providing the modelling and analysis tool HIT.

Contents

1 Introduction **1**

1.1 Instant messaging & presence architectures 2

1.2 Scalability analysis 3

1.3 Outline of the thesis 4

2 Definitions and State of the Art **5**

2.1 Architectures 5

2.2 Scalability 6

2.3 Instant Messaging & Presence 7

2.4 Database scalability 8

2.5 Connectivity 12

2.6 Instant Messaging & Presence Architectures 12

2.6.1 Jabber XCP 13

2.6.2 Facebook chat 14

2.6.3 Windows Live Messenger 15

3 Hyves Instant Messaging & Presence architecture **19**

3.1 General information 19

3.2 External services 20

3.3 Architecture overview 21

3.4 Extensible Messaging & Presence Protocol 23

3.5 Anatomy of an instant messaging & presence session 23

3.6 Functional description of the application node 24

3.7 Scalability problems 26

4 Modelling & Analysis Approach **27**
4.1 Queueing networks . . . 27
4.2 HIT: The Hierarchical Evaluation Tool . . . 30
4.3 Modelling approach . . . 35
4.3.1 Workload . . . 36
4.3.2 Modelling assumptions . . . 37
4.3.3 Scalability modelling . . . 39
4.3.4 Scalability aspects . . . 40
4.4 Analysis Approach . . . 41
4.4.1 Throughput analysis . . . 42
4.4.2 Response time analysis . . . 42
4.4.3 Number of machines . . . 43
4.5 Model of the current architecture . . . 45
4.5.1 Model . . . 45
4.5.2 Parametrisation . . . 48
4.5.3 Analysis . . . 50

5 Architecture Proposals & Analysis **57**
5.1 Discarded approaches . . . 58
5.2 Architecture 1: Evolutionary partitioning . . . 59
5.2.1 Overview . . . 60
5.2.2 Scalibility and remaining bottlenecks . . . 60
5.2.3 HIT model . . . 61
5.2.4 Analysis . . . 62
5.3 Architecture 2: Aggregated, batched presence updates . . . 69
5.3.1 Overview . . . 69
5.3.2 Scalability and remaining bottlenecks . . . 70
5.3.3 HIT model . . . 71
5.3.4 Analysis . . . 74
5.4 Architecture 3: Presence subscriptions . . . 78
5.4.1 Overview . . . 78
5.4.2 Scalability and remaining bottlenecks . . . 79
5.4.3 HIT model . . . 80

5.4.4 Analysis 81
5.5 Comparative analysis 86
5.5.1 Latency of message and presence propagation 86
5.5.2 Number of machines 87

6 Conclusion **93**
6.1 Related work 95

References **97**

1

Introduction

The popularity of an online service depends on many factors, such as ease of use, uniqueness, usefulness and a little bit of luck. But no matter how many people are willing to use a service, it is not going to be a success when the architecture is not able to handle the load. An example of a service that had to spend a lot of effort improving their architecture after it became popular is Twitter (Reisinger, 2008). Fortunately, they were able to adjust their architecture and are still successful.

Hyves is offering a social network service to Dutch users since 2004 and has been very successfull, growing each month in terms of users and load on the various systems supporting the site. Especially in the earlier years, the performance of Hyves was below users' expectations, causing complaints and a negative sympathy towards Hyves. While performance is good at present, Hyves would like to have plans available to avoid future performance problems caused by a load increase. The load can increase by an increased number of users and increased activity of existing users.

In this research, we design a scalable architecture for the Instant Messaging & Presence (IM&P) service offered by Hyves. Instant messaging allows users who are simultaneously connected to the service to exchange messages that are immediately delivered to the recipient. Presence is the information about the state of a user. It can contain a variety of information: whether the user is connected or not and its state, mood or location. Changes in the presence are distributed to the contacts of the user that has set these contact relations previously. The current architecture has been in use since the launch of the service in 2006, but is expected to not handle a large increase in number of users or more active users. The current architecture is not scalable: it does

not allow to add more hardware to serve higher load with the same level of performance per user.

We analyse the scalability of an architecture by measuring the behaviour of the architecture when the workload changes. The response of the architecture is measured by analysing the throughput of the network links and databases in the architecture. When more machines are added to an architecture and the workload is increased in a proportional way, the throughput of resources in a scalable architecture remain the same.

We use the Hierarchical Evaluation Tool (HIT) to create and analyse IM&P architectures. HIT translates these models to an open queueing network, which have been used to model computer architectures for a long time. It is possible to analyse end-to-end delay, buffer management and flow control in computer networks using queueing networks (Wong, 1978). More recent research showed that queueing networks can also be used to analyse higher layers in the network stack: a CORBA remote procedure call system was modelled and analysed to determine that it represents the real system accurately (Harkema et al., 2004). A web server architecture was modelled using queueing networks in (Slothouber, 1995). The model was used to analyse the effects of various adjustments to the architecture.

The aim of this research is to determine a scalable architecture for an instant messaging & presence service. We take a modelling and analysis approach to find the bottlenecks of this architecture as well as the bottlenecks of alternative architectures. Using the results, we recommend several architectures that provide better scalability than the current one.

In addition to the contribution of finding a scalable IM&P architecture, the method used to analyse scalability should be applied to other architectures as well.

1.1 Instant messaging & presence architectures

The world IM&P market is shared by a limited number of commercial services. The major internet technology companies Microsoft, Yahoo!, America Online, Facebook and Google each have a large instant messaging network. The architectures and protocols are proprietary. While the protocols can be reverse-engineered using packet inspection, finding out the architecture requires information from the companies themselves.

Only Facebook and Microsoft do provide this, and we use this information to develop alternative architectures.

The Facebook architecture takes a partitioned approach to distribute messages, but centralises the storage of presence information. Presence updates are transported in batches and the size is only a couple of bits per user. As the size of the presence information is a fundamental limit of this architecture, it does not support extended presence information, such as location information. In addition to that, the delay between a presence update and the reception of the update by other clients can be up to several minutes.

The architecture of Microsoft's Windows Live Messenger uses subscriptions to distribute presence. Presence information is stored on partitioned presence servers (PSs). Clients connect to connection servers (CSs) which send a subscription notification to each PS that stores presence of a contact. When that contact updates his presence information, the update is forwarded to the subscribed CSs, which transmit it to the clients.

1.2 Scalability analysis

The main goal of scalability analysis is to compare an architecture under different workloads. The performance of an architecture for a single workload is not an indication of the scalability, but comparing the performance of the architecture for different workloads determines if the architecture scales or not. For this comparison it is not necessary to know each performance aspect in detail, as constant performance factors do not contribute to comparisons. The aspects of the architectures that we do model are the network links and the databases. The effect that changing workloads have on these resources is used to determine whether an architecture scales or not. When an increased workload and a proportionally increased number of machines results in equal load on each resource for both the old and the new situation, an architecture is scalable.

HIT is a complete modelling package that provides the HISLANG modelling language, a modelling approach based on actions, components and services, the graphical modelling representation HITGRAPHIC and a set of model solvers. The action-based approach distinguishes HIT from state-based approaches such as petri nets and aligns better with IM&P architectures, where actions determine the load on the system. A

HIT model consists of *components* offering *services* to each other. This also conforms to architectures where components (application nodes, database nodes) offer services to other components and the user.

1.3 Outline of the thesis

The structure of this thesis is as follows. We start by explaining the terms scalability, architectures and instant messaging & presence in Chapter 2. The remainder of this chapter describes the state of the art for instant messaging & presence architectures.

In Chapter 3 we explore the Hyves instant messaging & presence architecture. We describe in detail the components in this architecture and the procedures that allow it to provide the instant messaging & presence service. After that, we give an overview of the bottlenecks as they were perceived at the start of this research.

Chapter 4 introduces the modelling techniques used for architecture analysis. We explain queueing network theory, the modelling tool HIT and our approach to model and analyse architectures for scalability properties. We apply this approach to the current architecture to study the bottlenecks.

In Chapter 5 we introduce three proposals for new instant messaging & presence architectures. Each architecture will be explained in detail, modelled and analysed individually for its scalability properties. After that, we compare the architectures by analysing latency properties and the relation between workload and the number of machines.

The analysis results are used in Chapter 6 to draw conclusions and reach the research goals as stated in this section. Furthermore, we give an overview of related research and future work.

2

Definitions and State of the Art

In this chapter we define the terms architecture, scalability, instant messaging and presence. After that we explain scalability solutions for databases and network connections. The chapter concludes by describing existing Instant Messaging & Presence architectures.

2.1 Architectures

An architecture is the description of the components of a system and their relations. A survey on the different meanings of architecture in the computer industry revealed that there are at least four groups of architecture users in practice. These are not mutually exclusive, but describe the way people use architectures as part of their work (Smolander, 2002).

- *Architecture as blueprint.* Architecture describes a system before it is built.
- *Architecture as literature.* Architecture is used as documentation of a system for later reference. This documentation can be produced in later stages of development, when all decisions have been made.
- *Architecture as language.* Architecture is used to align concepts of systems between people. Here, it is used throughout the development process.
- *Architecture as decision.* Architecture is used to make decisions on future implementations. This includes decisions on needed human resources, skills, money and

> time. The architecture can be used to make trade-offs between these conflicting factors.

In this thesis, we use architectures as a blueprint for a new system, but at the same time we use the architecture of the current system as documentation. Making a decision on the best architecture is the goal of this research as a whole.

To us, an architecture describes a system in terms of its components, relations and behaviour.

2.2 Scalability

Scalability is another concept for which multiple definitions can be given. The need for a proper definition was expressed at the end of the monolithic mainframe era (Hill, 1990):

> ... either rigorously define scalability or stop using it to describe systems.

As this quote suggests, the paper does not conclude with a definition. Only an attempt to link scalability with the mathematical definition of speedup and efficiency is given. This definition compares the execution time x of a task on one processor with the execution time on n processors. The ratio of these execution times is called the speedup. For scalable systems, the speedup must be equal to n.

A very broad definition of scalable systems regards scalable systems to be "economically deployable in a wide range of sizes and configurations" (Jogalekar and Woodside, 2000). A literature research revealed that there are at least four general types of scalability (Bondi, 2000).

- *Load scalability.* Load scalability means that the system performance remains acceptable while the load is increased. A reason for failing to be load scalable is that the system does not use the available resources, such as processing capacity. If in the end a system is not able to support an increasing load by utilising proportionally increased amount of resources, it is not load scalable.

- *Space scalability.* A system is space scalable if the memory requirements increase at most sublinearly when the number of items stored increases.

- *Space-time scalability.* If the system is able to perform properly when the number of items available increases, it is space-time scalable. An example is a search engine that returns results within the same average time when the amount of data is increased by an order of magnitude.

- *Structural scalability.* Structural scalability refers to calculating the impact of built-in limitations in the used components and assessing if those limitations impact the scalability of the system as a whole. Limitations on address space embedded in a protocol or hardware architecture are examples of causes for structural scalability.

Bondi uses the term performance in several locations. This is not a coincidence, as there is a close relationship between the performance and scalability (Haines, 2006).

> The terms "performance" and "scalability" are commonly used interchangeably, but the two are distinct: performance measures the speed with which a single request can be executed, while scalability measures the ability of a request to maintain its performance under increasing load. For example, the performance of a request may be reported as generating a valid response within three seconds, but the scalability of the request measures the request's ability to maintain that three-second response time as the user load increases.

In this thesis we use the term scalable to indicate an architecture where changing the amount of hardware resources allows the system to handle a proportionally changing workload while the performance remains the same. Scalability indicates the extent to which an architecture is scalable. We regard scalability as a property of an architecture.

2.3 Instant Messaging & Presence

Instant Messaging (IM) allows users to exchange messages that are delivered synchronously. As long as the recipient is connected to the service, the message will be pushed to it directly. This can either be realised using a centralised server or peer-to-peer connections between each client. Presence describes the state of a user. Possible

pieces of presence information are availability, status, location and mood. Using statusses such as *busy*, *on the phone* and *out for lunch*, users can indicate if they are willing to receive messages or the duration of their unavailability. Users can distribute this information to other users using a presence service, which can, again, be centralised or peer-to-peer. In this thesis, we will only discuss centralised instant messaging & presence (IM&P) services. Usually, users have to setup a mutual relationship before they receive updates of each others' presence. We use the term *contact list* for the group of users that are related to one user. In the literature, *friends list* or *roster* are common synonyms. The combination of instant messaging and presence allows users to determine if their contacts are willing to accept messages and to send them a message when this is the case.

Recent research on presence is aimed at adding context-awareness by deducing presence ubiquitously from the environment. Using fixed beacons with a known location, mobile devices can deduce their location and update the presence information of their user automatically (Peddemors et al., 2003). Taking such research into account, it can be expected that presence information will become much richer and granular in the future. Presence information will be updated automatically and more frequently, causing a higher load on the presence service.

2.4 Database scalability

An important component of the IM&P architectures discussed in this thesis is the database. In this section we describe how Consistent Hashing and partitioning can help creating scalable databases. At Hyves MySQL is used, but this section applies to database software in general. We introduce this topic by describing generic properties of databases.

The core function of a database is to store data and allow operations on that data, such as reading, adding, updating and deleting data. Operations can be grouped in transactions, where the whole transaction either fails or completes. The four key qualities of transaction processing are: Atomicity, Consistency, Isolation and Durability (ACID) (Haerder and Reuter, 1983). Of these properties, Consistency is the most important for scalability. Consistency ensures that the data is legal before and after a

transaction. The database administrator may define rules that the data has to comply to. During transactions these may be violated, but a transaction can only end successfully when the data is consistent.

To use databases in a scalable architecture, they must be scalable themselves. Load scalability and space scalability can be achieved by distributing all data over multiple hosts. While in it might be possible satisfy most capacity needs by adding equipment to a single machine, it is more economical to add more low cost machines and distribute the load (Barroso et al., 2003). Maintaining consistency in a distributed database system is very difficult, but possible using protocols such as Two Phase Commit (2PC) (Skeen and Stonebraker, 1983). Using this protocol each host first applies the transaction locally and reports to a centralised coordinator if it was successful or not. When all hosts have reported a success, the coordinator signals each host that it can finish the transaction. Otherwise, each host reverts the changes. 2PC only maintains consistency when all partitions are online, otherwise writing is not possible and the system becomes unavailable.

The relation between consistency, system availability and tolerance to network partitioning is known as the CAP theorem which states that only two of those can be achieved at any given time (Gilbert and Lynch, 2002). As the probability of a network or hardware failure increases for larger distributed systems, tolerance to network partitioning is a must. This leaves the choice between consistent data and availability of the system as a whole. In certain systems, such as financial or military systems, consistency might be of such an importance that it is better to have a general failure than to have an inconsistent data item, even if the inconsistency is temporary. However, in other systems, such as IM&P architectures, system availability is more important than consistency. Rather than abandoning consistency entirely, the strict ACID property can replaced by a more relaxed consistency requirement such as eventual consistency. This form of weak consistency guarantees that all readers of a data item will eventually read the updated value, if no other updates are made. The time between the update and the correct read is the inconsistency window. For eventual consistency, this window is determined by communication delays, load on the system and the number of replicas to be updated (Vogels, 2009).

Distributing all data items over all machines increases the read capacity of a database, as database clients can choose any machine to read from. This scenario is known as

full replication, as it requires replicating updates to each data item to be replicated to all machines. This makes writing resource consuming for both the network components and processing capacity on each machine.s In MySQL, full replication is known as *master-slave replication.* Clients send updates to the master machine which replicates this update to all slaves. Both the slaves and the master can be used for reading, but writing to a slave machine results in inconsistency, as this update is never transmitted to other machines.

To reliably increase both write and read capacity, the requirement of *full replication* must be dropped in favour of lesser degrees of replication. This means that some data items are copied to just a part of all machines. The most extreme degree is *no replication* where one data item is only available on one machine in the system. A *partitioning scheme* determines how data items are distributed over the machines. To distinguish between the physical term *machine* and the logical set of data it contains, we use the term *partition* to indicate a part of the data set that is stored together. *Shards* and *sharding* are common synonyms for *partitions* and *partitioning.*

A couple of important points have to be regarded when choosing a partitioning scheme.

- *Location lookup.* It should be straightforward to determine a physical location of a particular data item.
- *Partition modifications.* When applying partitioning to increase scalability, it should be easy to add partitions to increase capacity. Adding a partition should not decrease performance or harm the operation of the database.
- *Query limitations.* Complex queries can no longer be executed, as data is distributed over a large number of hosts. Joins and aggregations over the complete data set have to be split on all machines and aggregated at a central location.
- *Connection management.* Instead of one connection to one database containing all data, clients now have to create a connection to each of the partitions. Both adding more clients and adding more partitions cause the number of connections to increase. This can result in a cubic growth of the number of connections.

A partitioning scheme that excels at these points is Consistent Hashing. To explain the merits of this scheme, we compare it to the most naive approach, modulo hashing. When x items have to be partitioned over n partitions, and each item is identified by some unique integer, the partition to which item i is partitioned is $i \bmod n$. The advantage is that computing the partition is very easy. The disadvantage is that adding or removing a partition (changing n) affects the location of almost all data items. The outcome of the module formula is changed for all $i > n$, making partition modifications very disruptive.

Using this Consistent Hashing, adding a new partition affects only the data items that would actually have to be relocated to this new partition (Karger et al., 1997, 1999).

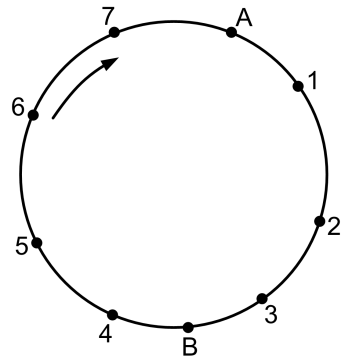


Figure 2.1: Consistent Hashing Continuum

Figure 2.1 shows the main concept of Consistent Hashing: the continuum. The continuum is a circle of a certain length on which both partitions and items are hashed. The length of the continuum must be much larger than the number of partitions. Each item is stored on the partition that is closest to the item on the continuum. On the figure, seven items ($1 \ldots 7$) are stored on the two partitions A and B. The items and partitions are hashed to a position on the continuum using some hashing function. Modulo hashing could be used, but schemes such as CRC32 and MD5 are also common. The result of this hashing is that items $1 \ldots 3$ are stored on partition B and items $4 \ldots 7$ are stored on partition A.

When a partition is added the key difference between modulo hashing and consistent hashing appears. Consider the addition of a new partition C that happens to be hashed

to a position somewhere between items 5 and 6. The effect of this operation is that only items 4 and 5 move while all other items remain at their original partition.

The procedure sketched above does not solve the problem entirely: adding a new partition only relieves one partition, while it should relief all partitions. A relatively small addition to the hashing scheme handles this: instead of representing a partition by one point on the continuum, N hashing functions map the partition to N positions. Adding a partition gives N new positions and relieves n partitions for $n < N$ and N otherwise.

The techniques described in this section allow architects to split data across different machines, lookup data using a decentralised algorithm and add or remove machines without harming the rest of the architecture.

2.5 Connectivity

Traditional network programming involves creating a multithreaded application that handles the connection to one client using one thread. The processing overhead of threading prohibits handling thousands of concurrent connections on one machine.

Using mechanisms such as `poll()` or `select()` handling many connections in one thread is possible. However, the processing complexity of these mechanisms is $O(n)$ in the number of observed connections. Again, this is prohibitive when handling thousands of connections.

Event-based connection handling libraries such as *libevent* are the solution to this problem. With an event-based approach, the application only needs to give a list of connections once and is notified when an event occurs one of them. Examples of events are incoming data and disconnections (Kegel, 2006).

These libraries were introduced around 2000 and have made the development of large scale network software possible. They allow machines of an Instant Messaging & Presence service to maintain lots of low-activity client-server connections.

2.6 Instant Messaging & Presence Architectures

While the architectures of most existing large-scale Instant Messaging & Presence services are undisclosed, some companies have given insight into the architecture of their

services. This section summarises the highlights of these architectures and discusses advantages and disadvantages.

2.6.1 Jabber XCP

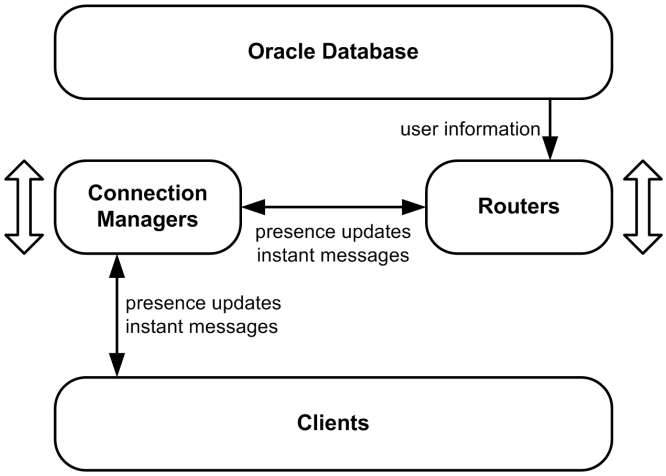


Figure 2.2: Jabber XCP architecture

The Jabber Extensible Communications Platform (Jabber XCP) is a commercial XMPP server, created by the Cisco subsidiary Jabber Inc. In association with Sun Microsystems a white paper was released describing an architecture supporting up to one million concurrent users (Jabber, 2007).

Figure 2.2 summarises this architecture. The white arrows indicate that the numbers of *connection managers* and *routers* are flexible: they can be scaled.

Clients connect to one of the connection managers. In the scenario described in the white paper there are two of these nodes. Incoming messages are forwarded to a router, of which six are used. These nodes route messages between connection managers and manage session creation. Each router connects to a single Oracle database that stores user information, contact lists and offline messages.

The authors of the white paper demonstrated the scalability of this architecture using a simulation. The connection manager, router and database nodes were deployed on eight core Sun T2000 1.2 Ghz machines. The database was stored on a Sun StorageTek 3510 FC array. Sixteen AMD dual core machines were used as client systems.

The connection managers and routers scale with the number of connected clients. Each connection manager maintained little over 500,000 connections. One router handles 420,000 users, but adding another router adds only about 150,000 connections. However, that number remains constant for each added router after the second. The white paper mentions that the database server is able to cope with the provided load easily and does not give scalability options for this component.

2.6.2 Facebook chat

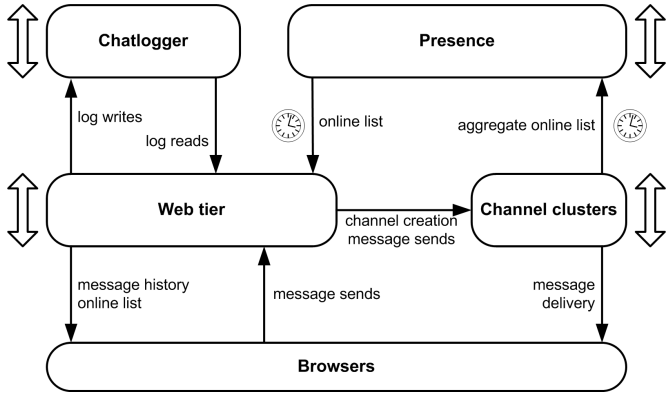


Figure 2.3: Facebook chat architecture

Facebook is a highly popular social networking website. Just like Hyves, it offers a chat service that is integrated in the website. It has disclosed their architecture in weblog posts and a presentation. According to their own reports, the architecture handles a peak load of more than four million connections. Over 300 million messages are sent per day and the architecture is using at least one hundred machines (Letucky, 2008; Piro, 2009).

The architecture of Facebook chat is given in Figure 2.3. The client of the service is code running inside the *web browser*. Instant messages are sent to one of the *web servers* in the *web tier*, which perform authentication and verify if the user is allowed to send messages to this destination. Messages are stored on partitioned *channel clusters*, such that each channel cluster is responsible for a portion of the users. The web server sends the incoming message to the channel cluster of the recipient. If the recipient is online, it is has a permanent connection to its channel cluster and receives the message

immediately. Otherwise, the web server receives an error message from the channel cluster. Recent conversations are stored temporarily at *chatloggers*. This allows the chat client to maintain a conversation history between web page reloads.

The only presence information supported by Facebook is an indication if the user is connected to the service or not. More granular information, such as being busy or temporarily away is not supported. Hence, presence updates are only sent when users log in or out. The updates are sent to the web server, which forwards it to the channel cluster that belongs to this user. The channel cluster maintains an list of the status of all its users and sends this list every thirty seconds to each *presence* server. The small amount of data makes it possible to have each presence server store presence for all users. Clients poll one of the web servers regularly to fetch the list of online contacts. Web servers retrieve this information from one of the presence servers. The polling rate is dependant on the level of user activity, but is at least a couple of times lower than the rate by which channel clusters forward information to presence servers.

While the Hyves IM&P client is also integrated in the website, it does not have deliberate delays in presence propagation, nor does any other major IM&P service. However, no complaints of Facebook users could be found on this topic, and it might be the case that they accept these delays.

2.6.3 Windows Live Messenger

Windows Live Messenger (WLM) is a stand-alone Instant Messaging & Presence application developed by Microsoft. It was released in 1999 as "MSN Messenger". The architecture behind this service was described in a video made in 2006, when 240 million people were using it. WLM uses the proprietary MicroSoft Notification Protocol (MSNP). The architecture described in this section is based used for version 16 of this protocol (Torre, 2006).

Figure 2.4 gives an overview of the WLM architecture. Before connecting to the main service, clients authenticate using the Microsoft Passport *authentication service*. Upon success, they receive an authentication token that can be used to connect to other services. Next, clients fetch their contact list from the *Address Book* service. Finally, they connect to one of the *Connection Servers* (CS) using the authentication token. They submit their contact list to the CS and remain connected thoughout their session.

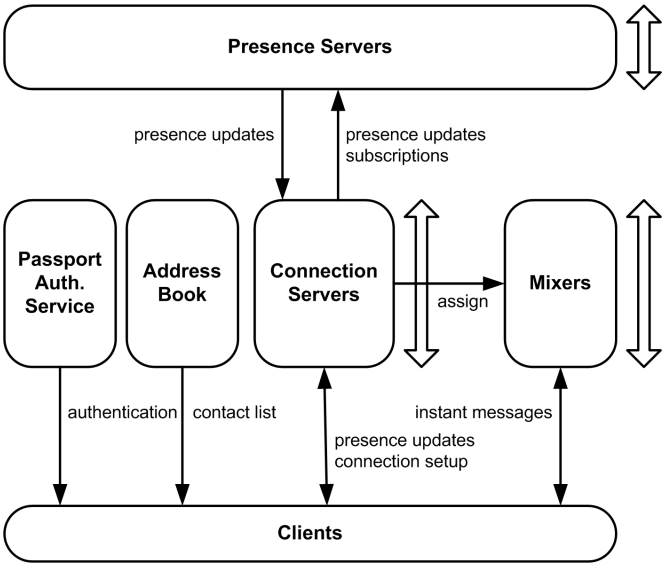


Figure 2.4: Windows Live Messenger architecture

Presence information is partitioned over "a lot" of *Presence Servers* (PS). Each user is assigned to one PS, which stores status information, a status message and a reference to a personal image. When the CS receives the contact list, it subscribes to presence updates for each person in the list by contacting the corresponding PSs. Clients submit presence updates to the CS, which forwards it to the right PS. The PS transmits the update to the CSs of subscribed users. Finally, these CSs transmit the updates to their clients.

Messages are exchanged using *Mixer* servers. Clients request a chat session from the CS, which is responsible for assigning a Mixer. The client connects to this Mixer and informs it of the other participants of the chat session. The Mixer invites the participants using their CSs. Due to firewall and home router limitations, the Mixer can't connect to the participants directly. All instant messages pass through just the Mixer from that point on, which relieves the CSs from maintaining chat sessions.

When the users are distributed randomly over all CSs, each CS will have a connection to all PSs. As long as the aggregate traffic over these links does not increase, this should be possible using for instance the connection handling technique described in Section 2.5.

An advantage of this architecture is that each component has a very specific purpose.

Each function of the system can be scaled independently. All Hyves IM&P functionality is offered with the same quality of service by Windows Live Messenger, in contrast to the Facebook architecture, which propagates presence updates with a delay. In addition to that, Windows Live Messenger has features that Hyves does not have, such as voice and video messaging, file transfer and integration with Yahoo! Messenger.

3

Hyves Instant Messaging & Presence architecture

Hyves is the most popular social network in The Netherlands. It provides users with traditional social networking features such as setting up a personal profile page and building relations to other users, as well as more comprehensive features, of which Instant Messaging & Presence (IM&P) are two.

We start this chapter by describing the environment in which the IM&P architecture operates. Next, we give an overview of the architecture and the XMPP protocol. After that, we describe in depth the interactions between the client and the architecture and the processes inside the architecture. We conclude by detailing the bottlenecks as they are currently perceived by Hyves engineers.

3.1 General information

Hyves offers variety of services 8,5 million subscribers inspired by the motto *always in touch with your friends*. The most important are features common to most social networks: users have the possibility to enter as much personal details as they like and they are shown on a public profile page. Users can invite others to their friends list, which is displayed prominently on the profile page. This friendship relation can be used to limit the visibility of the personal details to just the friends or friends of friends. For instance, a user may publish their date of birth to all visitors, but their telephone number to just their friends.

Hyves offers a couple of communication mechanisms. There is a private message system, allowing message exchange similar to web-based email. It is also possible to send public messages, called *scraps*, which are displayed to all visitors on the profile page. When both users are online at the same time, either using a stand alone client or on the website, they are able to exchange instant messages. Hyves users are also able to send each other email and SMS messages. Users can publish a various types of content to their profile page. They can, amongst others, write weblogs, conduct polls, upload pictures and videos and describe their state and location in a *who what where* message. Other users are able to comment on each of these content items individually.

A number of famous Dutch individuals or groups have created a Hyves profile to connect to their fans and supporters. This includes artists and other people famous from their show business career as well as politicians. While normal users are allowed to have only 1000 friendships, due to processing limitations, famous members can have an unlimited amount of friends. To limit the load on the system caused by these large numbers of friends, several features are disabled, amongst which is IM&P.

The IM&P service was added to Hyves in 2006. Since the beginning, the protocol has been the Extensible Messaging & Presence Protocol (XMPP) to leverage existing clients. While Hyves does offer a stand alone client, it does not prevent users from connecting with other XMPP clients. The service has used the same architecture since the launch of the service, although the number of machines used has increased from 11 to 37. All machines are of equal capacity. The server software is written in the programming language Python and uses the open source database server MySQL.

3.2 External services

The Hyves IM&P service is one of the many Hyves services, such as the personal profiles, social groups and a large number of external applications. These services share a lot of data to provide a consistent service to the user. The data about the users and their relations are not stored in the IM&P service itself, but requested from external services. These services are:

- *Contact List Service.* Contains information about the relations between users. In principle this relation is bidirectional, but the contact list service also contains

unidirectional relations. This service is read-only: relations are created using the site.

- *Authentication Service.* The authentication service allows the IM&P architecture to validate login information. It supplies a hash of the users' password, which is then compared to a hash of the input received from a user.

- *Notification Service.* The notification service uses the instant messaging service to deliver notifications. Instead of providing an interface to the instant messaging service, it calls an interface on the instant messaging service to deliver these messages. These messages relate to actions performed at the Hyves website, such as comments on someones profile.

3.3 Architecture overview

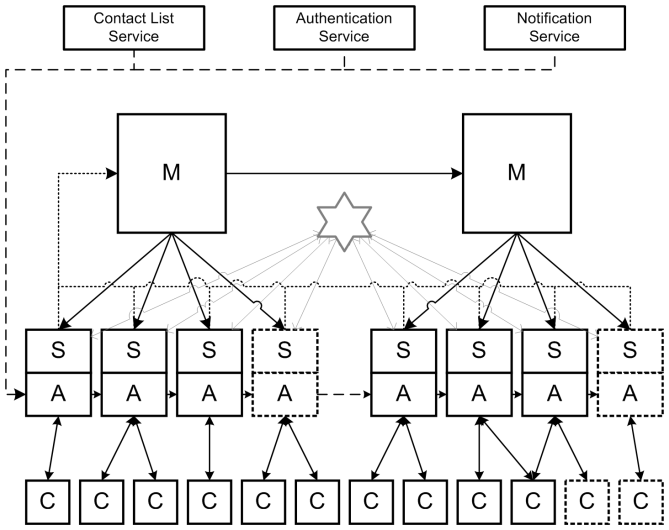


Figure 3.1: Overview of the current architecture

The current architecture is represented schematically in Figure 3.1. It shows that *application nodes* and *slave nodes* are related: physical machines contain both the application node and the database slave node to decrease look up times. The dotted borders of application nodes, slave nodes and clients indicate that the number of these nodes

can be changed. The grey star in the centre of the diagram visualises the connections between all application nodes.

Application nodes determine the behaviour of the system. They handle connections from clients, interpret requests, call external services or other nodes to fullfill these requests and create the response to be sent to the clients. Clients connect to one of the application nodes as determined by a weighted round robin scheme. The weight is manually set by the system operators to represent the relative capacity of each machine. Currently, all machines are equal and hence have equal weights. Messages and presence updates are submitted to this application node. Instant messages are sent from the originating application node to the application node to which the destination is connected. The mapping between users and application nodes is stored in the database. For every arriving instant message, the application node looks up this mapping to determine where the message should be routed to. For this look-up, no additional network traffic is required, as it is done on the local database slave node.

Presence updates are propagated similarly. When a user changes his presence information, the update is sent directly to the application nodes to which the contacts are connected. The difference between presence forwarding and message forwarding is that presence information is stored persistently. To achieve this, the application node forwards presence updates to the database *master node*. This node is connected to all slave nodes and replicates updates to these slave nodes.

The protocols used in this architecture are the MySQL protocol for data replication between master and slaves and for the traffic between the slave nodes and the application nodes (see Section 2.4) and the Extensible Messaging & Presence Protocol (XMPP) for the client-application traffic. The MySQL protocol is unicast, such that adding a slave causes additional outgoing traffic on the master node. The current architecture does not enforce a particular network layout. In practice, the machines are dispersed over various data centres and locations within those data centres.

The database of the current architecture stores user, presence and session information for each user that has ever connected to the service. The user information that is stored is just the username and the user identification number (userid). Other information is stored in external services.

The presence information consists of a status and a free text field. The status field contains one of the six statuses that users can set while they are online: online, busy, be

right back, away, on the phone and out for lunch. The free text field contains the entire protocol message of a presence update, as the XMPP standard requires implementations to maintain this information even when users are offline.

The session information includes an identification number of the application node to which a user is connected and the identification number of the connection within that application node. Empty values for these fields indicate that the user is not connected.

3.4 Extensible Messaging & Presence Protocol

The Extensible Messaging & Presence Protocol (XMPP) is an open standard for instant messaging & presence. It is published in Request For Comments (RFCs) by the Internet Engineering Task Force (IETF) and can be used freely. The protocol messages are formatted using XML, which allowsextending the messages with additional XML-formatted information. It was previously known as Jabber (Saint-Andre, 2004a,b).

Although in principle a protocol does not dictate the underlying architecture, XMPP is a single-endpoint protocol. XMPP clients connect to one machine of the architecture and transmit and receive both instant messages and presence updates. In contrast, the Windows Live Messenger architecture (Section 2.4) provides separate endpoints for the contact list service, authentication service and instant messaging.

In XMPP, users are identified by a username and domain name. Users can be connected multiple times using the same username, which allows to have a IM&P connection at work while the client at home is still running. Each connection is identified by a unique *resource*, which is combined with the username and domain name to yield a unique Jabber Identifier (JID). For instance, `jorritschippers@hyves.nl/phone` indicates a connection to the Hyves XMPP service from a telephone.

The domain part of the JID is designed to allow federation between XMPP services controlled by different organisations. At this moment, Hyves does not support this part of the XMPP specification, but Google Chat, a major public XMPP service, does.

3.5 Anatomy of an instant messaging & presence session

This section contains a detailed description of the interactions between an XMPP client and the Hyves XMPP server.

The client initiates a session by connecting to one of the application nodes. Clients can pick any application node available, but client software made by Hyves retrieves a list of active application nodes from the Hyves server management database and chooses one using a weighted round robin scheme. The server responds by sending the supported authentication mechanisms, after which the client picks one of them and the authentication is performed according to the mechanism. In practice, only DIGEST-MD5 is supported, which is a challenge/response mechanism. When the authentication is successful, the client proceeds by requesting the contact list, which at this point does not contain presence information, just their names and a profile image. The last initialisation action performed by the client is sending its own presence information. This is a trigger for the application node to send the complete presence information of the contacts to the client. This presence information is in principle indistinguishable from normal presence updates that occur during the session, but they are not displayed to the user during start-up time. Later presence updates cause the client to display a notification window.

When the user sends a message, a message protocol element is sent to the server. If the destination of the message happens to be connected to a different application node, the message is routed to the other node. Ultimately, the message is sent to the destination user. Presence updates are also sent to the application node, but now they are routed to all online contacts of the user. Additionally, the presence information is stored in the database.

During a session, a user might also receive notifications from other Hyves services. They are encoded the same as messages, but are tagged specially so the client software can display the message in a distinct way.

A user disconnects from the service by setting its presence status to *unavailable*. In that sense, most of the handling of this update is similar to the presence update action, except that the connection is closed after the sending that presence update.

3.6 Functional description of the application node

This section describes the instant messaging & presence service from the point of view of an application node.

Clients are distributed over the available application nodes using the aforementioned weighted round robin scheme. When a new connection request arrives, the server sends the supported authentication mechanism and waits for the application node. Upon confirmation, the application node generates a random challenge, on which the client responds with a hash in which the password is encoded. A password digest is requested from the Authentication Service and the application node calculates the same hash and checks whether the hashes are the same. When this succeeds, the application node checks if the client isn't listed in the blacklist and aborts the connection if so. After that, it creates a session in the master database node. When the client requests the contact list, the server requests the list of contacts from the Contact List Service, converts this list to an XMPP message and sends this message to the user. Finally, the client sends its own initial presence information. The application node sends this presence information to the first database master node and then retrieves presence information for all contacts from the local database slave node. The presence information is then sent to the client, finalising the login procedure.

When the user sends a message to another user, the application node uses the database slave node to determine the application node to which the destination is connected, if any. If the destination is online and connected to the same application node as the sender, the message can be delivered immediately. Otherwise, the message is forwarded to another application node over an internal network connection. Notifications are received from other Hyves services on an internal network connection and are routed similar to messages.

When a presence update arrives, the application node writes the new presence information to the master database node. The asynchronous replication process on the master node forwards the update to all other database nodes. The presence is also sent to all online contacts using the same routing technique as the messages. One message is sent to every user. It is important to note that the persistent presence storage is only there to provide initial presence information to newly connected users.

As explained in the previous section, a normal disconnection is simply a presence update with status *unavailable*. A premature disconnection is not noticed by the application node until it tries to deliver data over the TCP connection. Due to the nature of TCP, when there is no data to transmit, the connection will be entirely idle and a disconnection will not be noticed. The application node receives a transmission error

from the TCP stack when the client does not respond and updates the presence status to unavailable on behalf of the user. XMPP contains an optional ping feature that allows sending a heart beat signal at a regular interval, but the current application node implementation does not use it to detect disconnections.

3.7 Scalability problems

The database architecture used in the current instant messaging & presence architecture is known as the master-slave architecture (see Section 2.4). A known limitation in this architecture is the master, as it receives 100% of the write load of the system and is responsible for replicating the updates to all slaves. The outgoing network link is a bottleneck, as the number of packets is proportional to the number of slaves.

Increasing the number of slaves does not decrease the load of replicated update queries per slave, as each slave needs to process all updates. With an increasing write load, the resources left at the slave nodes to serve read requests become smaller and smaller, thus increasing the response time of the system (Nicola and Jarke, 2000).

This means that the current architecture is not scalable: the architecture is not able to increase its capacity linearly by adding more hardware.

4

Modelling & Analysis Approach

This chapter outlines the methods that we use to compare Instant Messaging & Presence (IM&P) architectures. As our approach relies on queueing network theory, this is explained first. Next, we describe HIT, the tool that we use to build and analyse models. After that, we discuss the specific modelling and analysis techniques used to compare architectures on scalability. We conclude this chapter by modelling the current IM&P architecture.

4.1 Queueing networks

A queueing network (QN) allows to model systems that process tasks using queues and processors (Trivedi, 2001). It consists of interconnected *queueing stations* that are composed of a *queue* and a *server*. Jobs (or customers) enter a queue, wait there until they are allowed to enter the server and leave the station after being processed by the server. In *closed* queueing networks, the number of jobs is fixed and jobs never leave the network. When a job leaves one queueing station, it always moves to another. In *open* queueing networks, jobs arrive from an outside source an may leave the network to an outside sink Here, the total number of jobs is infinite and the number of jobs in the network depends on the rates of incoming and outgoing jobs. In the following we deal with open QNs.

The following properties of a queueing station determine the behaviour.

- The *arrival process* of a queueing station gives the rate at which new jobs arrive at the queue. This rate is the sum of arrivals from outside the network and from

other queueing stations.

- The *service time distribution* of a queueing station determines the time it takes to process a job in the server. This does not include the waiting time in the queue.
- The *number of servers* in a queueing station determines the degree of parallelism of a queueing station. This number can be infinite, meaning that jobs never have to wait, as there is always a server available.
- The *capacity* of the queueing station determines the maximum number of jobs in the station. This includes both the waiting jobs and the jobs in the server. New jobs arriving at the queue are dropped when the queue is full.
- The *population* determines the total number of jobs. For open networks, this is infinite.
- The *scheduling discipline* decides which job from the queue is selected to receive service. Besides the default First-In-First-Out (FIFO), other common strategies are Last-In-First-Out (LIFO), Processor Sharing (PS), where all jobs are served simultaneously , but the total service rate is shared between all jobs, and Priority Scheduling (PRIO).

A shorthand notation to describe the properties of a queueing station is Kendall's notation (Kendall, 1953). A queueing station with Markovian arrivals and service times, one server, an infinite capacity and FIFO scheduling discipline in an open network is noted a $M|M|1|\infty|\infty|FIFO$. As *infinity* is the default for the capacity and population and $FIFO$ is the default scheduling discipline, this can be abbreviated to $M|M|1$.

When the service rate is lower than the arrival rate and the capacity and population are infinite, jobs will stay in the queue infinitely long.

- *Response time.* The time jobs spend in the station.
- *Throughput.* The rate at which jobs leave the station.
- *Population.* The number of jobs in a queueing station.
- *Utilisation.* Percentage of time that the server is servicing jobs.

For $M|M|1$ queueing stations, the utilisation ρ is equal to the arrival rate λ divided by the service rate μ. When λ exceeds μ ($\rho > 1$), jobs arrive faster than they are processed by the server, overloading the queue. In the opposite situation, the queue is stable and the response time is finite number. Queueing stations with a limited capacity will never overload, but jobs that arrive at a full queue will be lost.

The relation between the mean population L, the mean response time W and the mean arrival rate was proved by John D. C. Little and hence is called Little's Law (Little, 1961).

$$L = \lambda W$$

When a job leaves the server, it has spent on average W time units in the station. In that time, λ new jobs have arrived per time unit, such that there are now λW jobs in the station. This only holds for stable queues with unlimited capacity. Due to the use of mean values, this law is independent of the arrival process or service time distribution.

As these properties are related to random variables, their values at a specific time are difficult to compute. Instead, the mean values in the long run offer a good insight in the queue behaviour, while being computable relatively easily. These values do not always exist: in the case of overloaded queues, computation of mean values is not possible.

Many different arrival processes and service time distributions are available to a modeller. In Communications Science, Markovian or memory-less processes are preferred, where the future state only depends on the current state. Calculating properties of a station with this property is relatively easy, while the behaviour still matches reality. For continuous time processes (as opposed to discrete time processes), only Poisson processes have the memory-less property. In a Poisson process, the inter arrival times are negative exponentially distributed. The applicability of Poisson processes for arrivals at telecommunication systems was first discovered by Erlang (Erlang, 1909). Since then, many arrival processes in communications science are assumed to be Poisson processes.

4.2 HIT: The Hierarchical Evaluation Tool

The Hierarchical Evaluation Tool (HIT) evaluates queueing models written in the HIT Specification Language (HI-SLANG). These models can be partially constructed using a graphical notation in the supporting tool HITGRAPHIC. This section describes the functionality of HIT, the methods HIT uses to evaluate models and the advantages of using HIT (Beilner et al., 1989, 1994).

HIT allows the modeller to describe the system at different layers of abstraction. Each layer exposes functionality to upper layers and uses functionality of higher layers. The bottom layer is formed by the parts of the system that do not need any more refinement for the modelling task at hand. This makes HIT useful to model compositional systems. The system can be decomposed to the desired level of abstraction and analysis can be performed on each of the layers. At each layer, the functionality of the system is represented by one or more HIT *components*. All components and the relations between them form the hierarchy of the model. The functionality that each component provides to a higher layer is modelled as HIT *services* associated with a component. They are called from components in higher layers in the hierarchy and call on services of components in lower layers. Services spend modelling time according to a specified procedure.

One component forms the top layer of the hierarchy. This is the component that only used functionality provided by lower layers and is not used by other components. Naturally, this component represents the highest layer of abstraction for the system as a whole, and the services of this component represent the services that the system offers to the outside world. They are not called by other components, but are used as entry point for experiments. Executions of the model are started by creating HIT *processes* that call a service of the top level component at a specified rate. Depending on the implementation, this invokes services on lower layers of the hierarchy until a bottom service is called. The benefit of this approach is that components at intermediate levels can be regarded as top-level components for the remaining part of the hierarchy. This allows each component to be analysed separately within the same model.

The implementation of services is done using HI-SLANG. This language contains all common programming constructs such as variables, constants, branches and loops. Models, components and services can be parametrised to model different configurations

that only differ by some constant values. In this way, many different configurations can be run easily.

HIT allows the modeller to specify the performance measurements of interest. For each component, throughput, response time and utilisation can be measured during execution of the processes. Mean values and bounds are calculated and returned at the end of the execution.

HIT offers four methods to solve the models and calculate the performance measures (Büttner et al., 1999). The Dortmund Queueing Analyzer 4 (DOQ4) and the Linealizer 2 (LIN2) methods solve models analytically. The advantage is that solving models is fast, but the feature set of these solvers is limited as many HI-SLANG programming constructs are not available. DOQ4 uses mean value analys for state independent stations and a convolution algorithm for state dependent stations. The LIN2 solver calculates performance measures of large separable networks, including bounds on these measures. Slightly more features are available when solving models numerically using the Markov solver, which analyses general, non-separable networks. It generates a Markov chain for the model and the performance of this solver depends on the size of the state space. All features of HIT are available when using the simulative solver. The disadvantage is that simulation is very time-consuming and the results are only approximations. As we only use analytically solvable models, the remainder of this section only discusses features available for this class of models.

Upon execution, HIT translates the model into a queueing network. HIT processes are the customers or jobs of the queueing network. The time between the start of new processes needs to be negative exponentially distributed to create an analytically solvable Poisson process. HIT supports this by providing a negative exponentially distributed random variable according to a given rate.

Each component is translated into a series of queueing stations. HIT allows more precise approximation of the real world behaviour by implementing a component using multiple queueing stations instead of one. The flow of jobs between the queueing stations of one component is determined by the *control procedures*. The flow between queueing stations of different components is controlled by the service invocations of the models. Figure 4.1 shows the HITGRAPHIC representation of the queues and control procedures of one component.

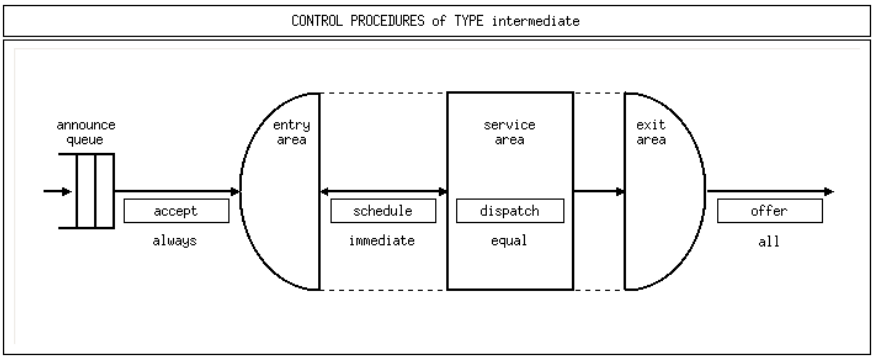


Figure 4.1: HITGRAPHIC Control Procedures overview

The first queue of a component is the *announce* queue. Every process waits in this queue for a time determined by the *accept* procedure. When a process is accepted into the component, it waits in the queue of the *entry area*. The *schedule strategy* determines which processes are moved to the *service area* queue to receive service. The *dispatch* strategy determines the rate at which processes leave the *service area* and move to the *exit area*. A process waits in this area for a time determined by the *offer* procedure. For analytically solvable models the *accept* and *offer* procedures are not applicable due too tool constraints. A comparison of all possible *schedule* and *dispatch* strategies is listed in Table 4.1.

The combination of the *immediate* schedule strategy with the *equal* dispatch strategy results in a queueing station that is known as Infinite Server with Kendall notation $M|M|\infty$, as all processes are serviced as if they are the only process in service. In contrast, the combination of *immediate* with the *shared* dispatch strategy is known as Processor Sharing ($M|M|1|\infty|\infty|PS$). The service rate per process decreases linearly with the number of processes in service.

HIT includes a number of predefined component types. The most basic component is the *server* component. This component can be configured with a base rate at which processes are given service. The amount of service that a process needs is given as a parameter to the *request* service. The rate and amount of service are unit-less; it is the task of the modeller to keep the units of the values consistent. The analytical solver requires the requested service to be negative exponentially distributed.

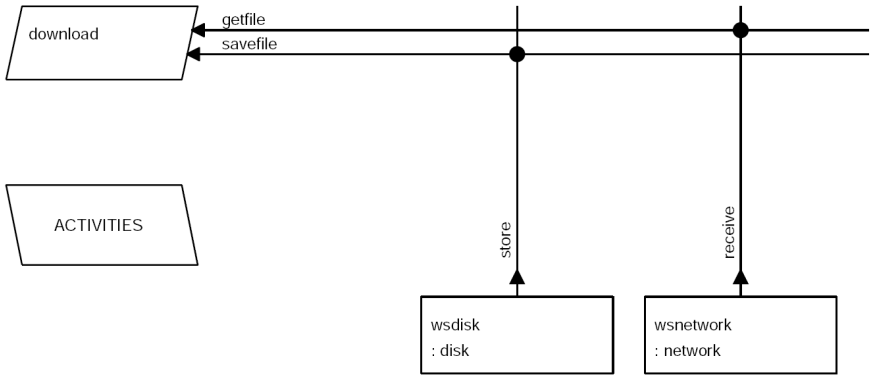
Schedule strategies	
immediate	All processes enter the *service area* immediately
fcfs	First come first scheduled: the oldest processes are serviced first
lcfspr	Last come first scheduled with preemptive resume: the newest process receives service immediately. The process currently in service moves back to the *entry area*.
random	A new process to receive service is selected randomly using a uniform distribution.
Dispatch strategies	
shared	Each process receives an equal share of the total service rate
equal	Each process receives the full service rate
sharedsd	State dependant variant of *shared* where the total service rate depends on the number of processes in service.
equalsd	State dependant variant of *equal* where the total service rate depends on the number of processes in service.

Table 4.1: Strategies for analytically solvable models

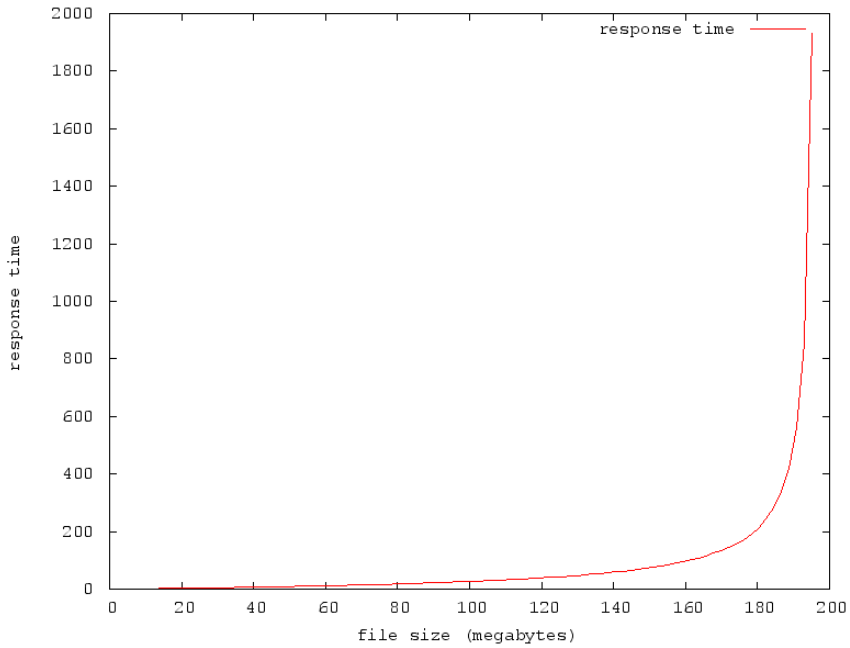
During our analysis of scalable architectures, some limitations of HIT became apparent. Creating an array of identical components is possible for all types of solvers, but accessing a random component in this array is only possible using simulation. The number of components in an array must be statically defined, making it impossible to modify this number between experiments. Simulation takes a long time, as expected, but the HIT process tends to crash for no apparent reason after a while.

Example: file download A simple HIT model for the task of downloading a file on a workstation is given in Figure 4.2(a). The figure gives the model of the *workstation* component. It uses the services of the *disk* and *network* components, displayed on the bottom of the figure. They expose the *store* and *recieve* services, respectively. The *download* service invokes the *recieve* and the *store* services.

The *network* component is modelled using the schedule strategy *immediate* and the dispatch strategy *shared*, as the network capacity is a shared resource. The speed of the network is 100 megabit per second. We take seconds as our measure of time for



(a) HITGRAPHIC representation



(b) Response time

Figure 4.2: HIT workstation example

this model and bytes as the measure of work. The service rate of this component is thus 13,107,200.

The *disk* serves just one job at a time and uses a first come, first served scheduler. This is modelled as a component with a *fcfs* schedule strategy and a *equal* dispatch strategy. Given a speed of 15 megabyte per second, the service rate of this component is 15,728,640. The *download* service consists of just two service invocations: first a file of a certain size is downloaded and then stored on the disk.

The following experiment repeats this process for file sizes ranging from 10 megabyte to 1 gigabyte. The analytical analyser determines the response time for this operation, given that for each file size the *download* service is executed once every fifteen seconds.

The *download* service of the *workstation* component is shown as a parallelogram on the left. Both underlying components, *disk* and *network* are instantiated as *wsdisk* and *wsnetwork*. Their services are shown as upwards outgoing arrows. These arrows are connected to the *download* service, indicating that the *download* service uses the *store* and *receive* services. The names *getfile* and *savefile* are the local names given to these services. Here, they are displayed in the order in which they are used. The order of invocation depends entirely on the HI-SLANG code associated with the *download* service.

This code passes the *filesize* parameter to both services that is set to a specific value by the experiment execution code at a higher layer.

```
getfile(filesize);
savefile(filesize);
```

Figure 4.2(b) shows the response time of the processes. It shows that the response time increases moderately until around 140 megabytes. After this value, the response time grows asymptotically until it reaches 185 megabytes. For larger file sizes, the response time is infinite.

4.3 Modelling approach

We model the current architecture and each alternative using the same approach. In this section we describe the workload, the assumptions that help abstract from unnecessary details and the types of measurements provided by the models.

4.3.1 Workload

In this research, we limit ourselves to the following actions that users can perform on the system:

- *log in*: logging in to the system, receiving the contact list and the initial presence information for all contacts. This action also broadcasts presence information to all contacts.
- *log out*: removing presence information for a user and notifying contacts of the logout.
- *send message*: send a single message from one user to another.
- *receive notification* from other parts of the Hyves architecture.
- *update presence*: broadcast a presence update to all contacts and store the presence information persistently.

To indicate the frequency of these actions, the current architecture is monitored. It maintains counters for every database statement issued to the database slave nodes and to the database master node. Using the source code of the application node, these statements can be correlated with the actions.

Figure 4.3 shows the user activity of a typical day. Statistics of other days show a similar pattern. The x-axis shows the time of the measurements, starting and ending with midnight. The y-axis shows the rate of each action in number of executions per second.

As can be seen, the number of login and logout actions is very close, which is expected, as every user logs out after a while. Another observation is that around noon, users are relatively idle: there are less messages exchanged than logins. In the evening people appear to chat more, as the curve of the number of messages is above the curve of the number of logins.

We will use the peak values of these measurements as input for our models. The combination of these values represents one unit of workload, such that an increase or decrease in workload is equal to an proportional increase or decrease for all values. The extraction of model parameters from the statistics is described in detail in Section 4.5.2.

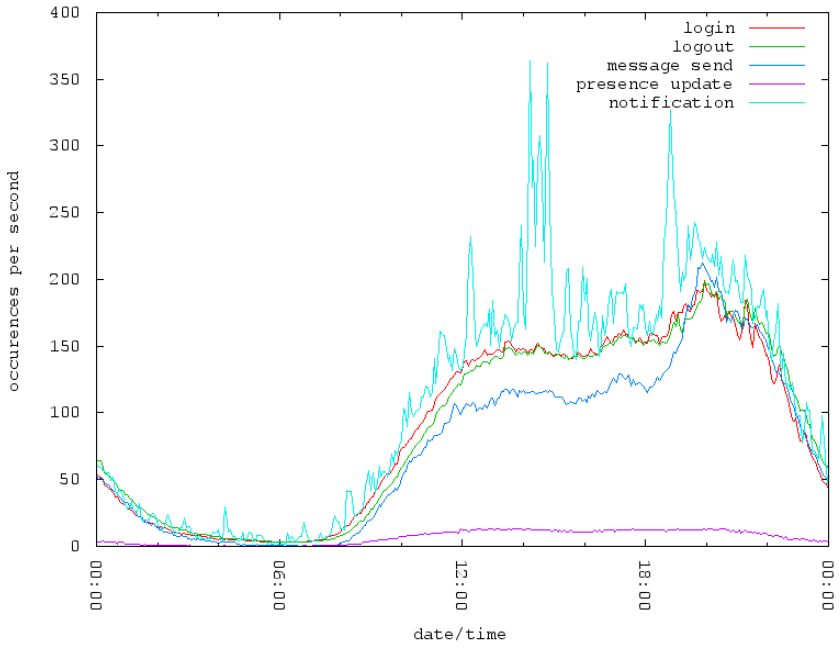


Figure 4.3: Activity in the current chat architecture of typical day

4.3.2 Modelling assumptions

The following assumptions are made to limit ourselves to the essential parts of this research, namely comparing architectures on scalability properties.

Assumption 1 There is no relation between de number of users and the workload.

Figure 4.4 shows the available data on workload and on the number of users for a considerable time period. The vertical bars show the number of users and the number of instant messages received per month. The line is derived from these two statistics and shows the number of messages received per user per month. As a result, the line shows the relation between the number of users and their activity. The graph shows that in the last three months this relation is constant, but fluctuates highly in 2008. The data do not indicate that there is a constant relation between number of users and the rates of their actions. Hence, we do not use the number of users as a workload parameter of the architecture models. Only the rates at which the various actions occur are used.

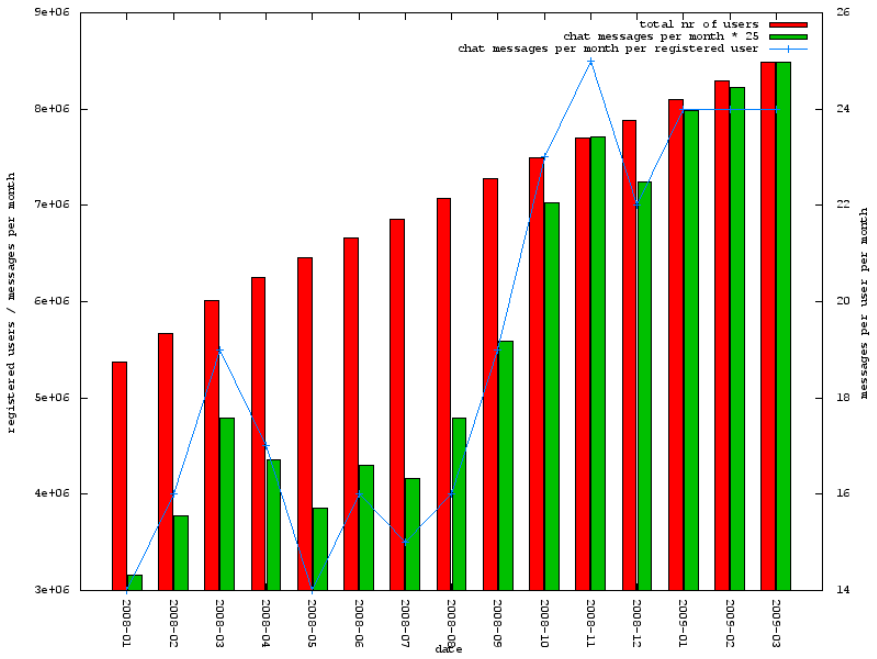


Figure 4.4: Relation between user base growth and messages received by users

Assumption 2 The average size of the contact list is independent of the number of registered users.

When the number of users is small, the availability of suitable users to establish a relation with is limited by this number. As more users register, the number of contacts per user reaches a level where additional registrations do not have influence on all but some of the existing users. For the numbers of registered users we investigate, we assume that they do not influence the average contact list size.

Assumption 3 An architecture that is able to serve a certain workload is also able to serve lower workloads.

As can be seen in Figure 4.3, peak loads are reached only once per day. By using the rates of the actions as workload and testing for rates of peak loads, the non-peak loads are not tested in the models. We have to assume that architectures are able to handle the full range of loads when they are able to handle the peak load. For the current architecture, this is true.

Assumption 4 The external Hyves services are scalable.

Recall from Section 3.2 that the Instant Messaging & Presence (IM&P) architecture communicates with a number of other services within Hyves. Higher workloads on the IM&P architecture cause higher loads on these services. We focus only on the scalability of the IM&P architecture itself and assume that these other services are able to handle this additional load.

Assumption 5 There is no feasible way to utilise grouping within the user set.

The set of relations between users can be represented as a graph with vertexes representing users and edges representing relations. Finding groups of users that have relations with all other users in the group equals finding cliques in the graph, a problem which is considered to be NP-complete (Abello et al., 1999). With 8,500,000 vertexes and between half a billion and one billion edges, finding partitions requires a lot of processing time. Furthermore, several hundreds of thousands of new vertexes and even more edges are added each month, which renders maintaining the partitions infeasible.

Assumption 6 Workload is perfectly balanced between components of the same type.

In our analysis of the architecture models we assume that the workload can be balanced perfectly over all components in each layer of the architecture. For example, if an architecture contains ten application nodes and one hundred login actions are performed, each application node needs to handle ten login actions. The statistics used to generate Figure 4.4 have shown that this assumption can be safely made. The assumption simplifies analysis, as only mean performance values for all nodes of a certain kind have to be calculated.

4.3.3 Scalability modelling

The term scalability relates to both the workload and the resources of a system. Hence, a scalability model captures the relation between workload and resources. We model both the workload and the resources using HIT (see Section 4.2). Combined with the architecture details, these aspects form the architecture model. It is split in three layers:

- *Architecture layer.* The architecture layer maps the actions to invocations on nodes of the architecture. At this level, the architecture is only represented on a high level that abstracts from the actual hardware components that support it.

- *Component layer.* On the component layer the high level actions are translated in actions on resources, such as network links and databases.

- *Resource layer.* The components defined at this layer represent the resources that support an architecture. The effect of high level actions on this layer determine the scalability properties of the architecture. At this layer, only the resources of interest for scalability analysis are modelled.

Using HIT, these layers can be combined to one model while the individual parts can each be modelled at convenient abstraction levels. At the upper level, parameters can be tweaked to model different user characteristics. This can be done to model a growth of the number of users or to model the change of behaviour of the average user. The output of the model contains the performance measures on resources, such as utilisation, throughput and response time.

HIT analyses the model by translating it to an open queueing network. The measures will be generated using the analytical DOQ4 solver.

4.3.4 Scalability aspects

It has been stated in the beginning of this section that modelling for scalability means to compare the response of the resources of an architecture to different workloads or to other models. In this research, we consider only the following resources of each architecture:

Database access Our experiments have shown that the raw database performance is much higher than the performance of the database access layer in the server software. Building the database request and interpreting the database response on the client side takes more time than retrieving the data from memory. This is related to the fact that databases in the current IM&P system are optimised such that every read or write access utilises indices. The table containing all presence information is loaded entirely into main memory, further lowering the processing time in this component.

This is modelled by ignoring the pure database access time. Real-world phenomena such as lock contention, replication delay and query caching are ignored as they only affect the performance of a single database, not the scalability of an architecture. Database access is modelled as a $M|M|\infty|\infty|\infty|PS$ queue, where read and write requests have different service demands. This results in a low delay that is independent of the number of processes accessing the database for low amounts of simultaneous queries, but rapidly increases when the load approaches some threshold. In this way, the delay is insignificant for a normal load, but will be prohibitive when the component is overloaded, indicating a lack of scalability.

Besides the processing time, the model also provides measures such as query rate. These measures can be used to determine the relation between workload and component behaviour and to determine the practicality of the architecture by human observation.

Network links The diagram of the current architecture (Figure 3.1) shows that the database master node is associated with many slave nodes. Incoming updates to the master database are translated in replication traffic between the master node and its associated slaves. This replication traffic is simply a duplicate of the incoming update statements: each statement is sent to each slave. At peak times, this can cause a high load on the outgoing network link, hence the links themselves are modelled and measured.

As explained in Section 2.5, the number of connections per link is virtually unlimited, as long as the aggregate traffic per link is bound. Therefore we only model the incoming and outgoing traffic per network link.

4.4 Analysis Approach

In this section we present a number of approaches to assess scalability. The first approach analyses architectures individually by relating an increase in capacity and load with the throughput per node. The second approach gives the relation between the load and the total number of machines necessary to support that load.

4.4.1 Throughput analysis

We have seen that in the current architecture the number of slave nodes is variable. In new architectures, similar variables exist. In a scalable architecture, adding a certain factor of nodes should increase the capacity by that factor. We analyse the architectures to determine if this property holds by increasing the number of nodes and the workload proportionally and measuring the resulting load per node. Ideally, the load per node remains constant. If it increases, it means that the additional workload is not served just by the additional nodes, but causes an increased load on each machine. This indicates non-scalability. A decrease in the load per node indicates that the node is does not receive a proportional amount of the workload.

We apply this technique for each architecture and each adjustable node count variable within that architecture. We take the current peak workload rates as our basic unit of workload and increase that workload up until ten times the current rates. At the same time, we proportionally increase the number of nodes of one type, while the number of nodes of other types remain constant. From each measurement, we take the throughput at each resource in that node. Recall that resources are network links and databases. We analyse the throughput in each of these scenarios to determine scalability.

The workload parameters described so far are related to just the usage of the service and not to the total amount of users and active users. For some of the workload experiments, we double these parameters as a comparison.

4.4.2 Response time analysis

The Facebook chat service shows (see Section 2.6.2) that response time is not an issue for presence propagation. Instead, the delay for transferring instant messages must be low, as users might become dissatisfied with the service if the messages do not arrive immediately. Hence, we will analyse the response time of the *sendmessage* action for each architecture.

We need to use slightly modified versions of our models for this analysis. The analytically solvable models use a single queue for each group of nodes, of which the service rate is multiplied by the number of nodes it represents. For instance, the model of an architecture with six application nodes contains one application node component

with a service rate of six times the service rate of one application node. Solving models is easy in this way, as the analytical DOQ4 or LIN2 solvers can be used. However, they underestimate the response time compared to a queueing network with six separate queues (Haverkort, 1998). The adapted models can only be solved using simulation, yielding an approximation of the mean response time. The 95% confidence interval is calculated to indicate the accuracy of the estimation.

Message sending involves mainly the application nodes. Therefore, we fix the number of all other nodes and vary the number of application nodes from one to ten. At the same time, the workload is changed proportionally, similar to the previous experiment. Note that the latency measured by this analysis is not the latency as it is experienced by the end user. Only the databases and links are modelled, leaving out the processing time at each node. Also, the latency of the network links from the users to the architecture are not taken into account. However, the analysis does provide insight in the effect of scaling the architecture on the response time.

4.4.3 Number of machines

In this section we present an analysis method to analyse models of architectures on the relation between workload and machines. Using this method, not all performance aspects of an architecture have to be modelled to find this relation.

As explained in detail in Section 4.3.4, we only model the performance of network links and databases. Experiments have shown that these two resources contribute only for a small amount to the performance of the currently deployed architecture. The database is able to handle more queries than it currently does at peak workloads and the network links are far from being saturated. Instead, implementation details such as the programming language, data structures, algorithms, operating system and hardware platform determine the performance of the system. These details are difficult model for an implemented architecture and even more difficult to predict for non-implemented architectures. We use the following reasoning and assumptions to help us using these unknown performance measures to our advantage.

- Within Hyves, the current user-visible performance of the architecture is considered to be well within expectations. There are no complaints of delays, even at peak hours.

- We assume that whatever the exact utilisation of the current architecture is, it is at least sufficient to meet the users' expectations. A future architecture using roughly the same implementation details with the same utilisation is assumed to be sufficient as well.

- We assume that the utilisation of the resources of the architecture that we can model is proportional to the total utilisation of the architecture as a whole. Even if at peak load the models show that links and databases are utilised well below their individual maximum capacity, we assume that this utilisation represents the maximum utilisation of these resources.

- Given that the current number of machines is able to handle the current peak load, we assume that for each other architecture, that number of machines is able to handle the peak load.

Under these assumptions we can predict what the number of machines must be to provide the same level of service as now, but with another architecture and another workload. We now explain how this result is achieved in practice.

We need to know two properties of the current architecture: the number of machines and the peak workload w. By running the model of a new architecture with these parameters, we can retrieve the utilisation $u_{1,r}$ for each resource r. $u_{x,r}$ is the utilisation of resource r for workload $x * w$. Then, we increase the workload by a factor i. We use the model of the architecture to determine the minimum number of machines necessary to provide for an utilisation $u_{i,r}$ that is smaller or equal to $u_{1,r}$ for each r.

The machines used in the current architecture are split in two functional groups. Two of the 37 machines are used just for data replication, while the remaining 35 machines directly serve actions generated by users. Similarly, new architectures have multiple functional groups of machines. We need to find a way of determining the ideal distribution of machines of the groups.

The scalability model that was created using the guidelines of Section 4.3.3 has a clear distinction between these groups. The utilisation of a single group is given by the average utilisation of its resources. The utilisation of an architecture is given by the average utilisation of its groups, regardless of the number of resources in the group. In this way, groups with more resources are not overrepresented in the resulting average.

This aggregate utilisation is only used to determine the best distribution of the initial number of machines over the nodes. The utilisation per resource of this distribution is used as upper limit for this resource when measuring other workloads.

4.5 Model of the current architecture

4.5.1 Model

Based on the approach outlined in the previous section a model was developed using the graphical modelling tool HITGRAPHIC and the associated textual modelling language HI-SLANG (see section 4.2).

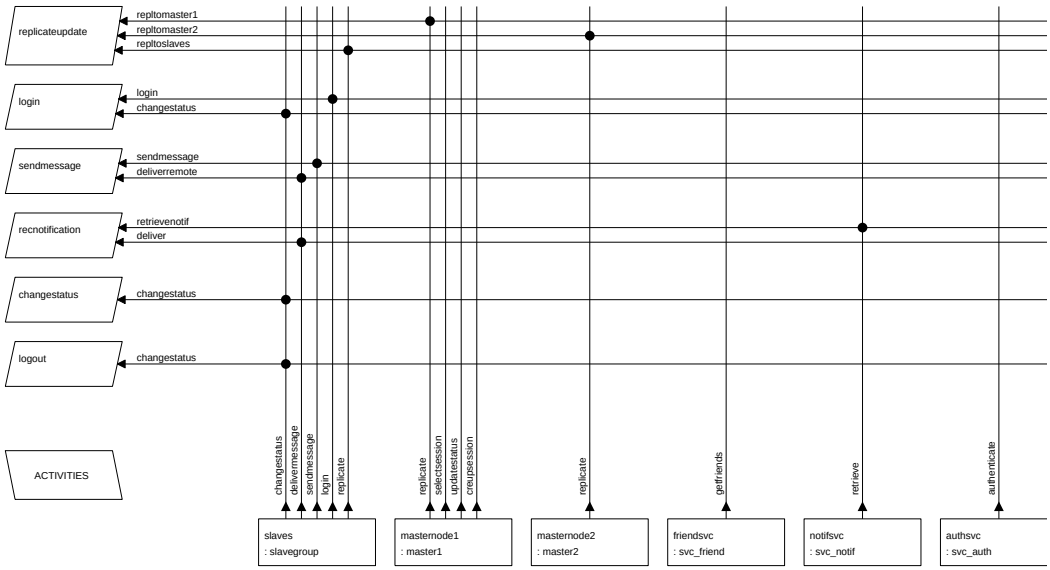


Figure 4.5: Top layer of model of current architecture

The architecture layer of the model is displayed graphically in Figure 4.5. The services of the architecture are displayed as parallelograms on the left side of the model. The model contains one service for each action. The bottom row of rectangles represent the underlying components of the architecture. The first three rectangles refer to the internal architecture components: slaves, master node 1 and master node 2. The remaining components refer to the external services.

The services of the architecture invoke services offered by the underlying components. Each used service is represented by a horizontal line originating at the service that uses an underlying service, such as *login*. Each offered service is represented by a vertical line originating at the corresponding component. The dot on the intersection indicates the relation between the used service and the offered service. The model shows that that the services *login*, *changestatus* and *logout* are similar, as they all use the service *changestatus* of the *slavegroup* component.

The *replicateupdate* service is not directly associated with an action. This service represents the asynchronous replication of presence updates from master nodes to slave nodes. In this way, replication delays do not lead to delays in the *login*, *changestatus* or *logout* processes.

Because of modelling constraints, the group of all slaves is modelled as one component. Arrays of identical components are possible in HIT, but utilising this functionality would require solving the model using simulation instead of analytical methods. The implementation of the services of the slave group takes into account the number of slaves represented by the group. The HI-SLANG code executes a certain operation n times when it needs to execute that operation on all n slaves.

Figure 4.6 shows the HITGRAPHIC model of the *slavegroup* component. Each of the high level services of the slave node is mapped to services of components. The most important components are the links and the local databases, as they determine the scalability properties. Components with a grey background are shared instances with other parts of the model, such that invocations on these components from anywhere in the model contribute to the same measurements.

Figure 4.7 shows the hierarchy of all components in the architecture model. It also shows the measurement points, or *evaluation objects* in HIT terminology, associated with each component. They generate measurements when the model is evaluated. The *links_receive* component of the *slavegroup* component contains three evaluation objects: one to measure all incoming traffic, one to measure incoming traffic from clients and other slave nodes and one to measure incoming traffic from one of the master nodes. This allows the incoming traffic from different sources to be analysed separately.

The parameters for the model are listed in Table 4.2.

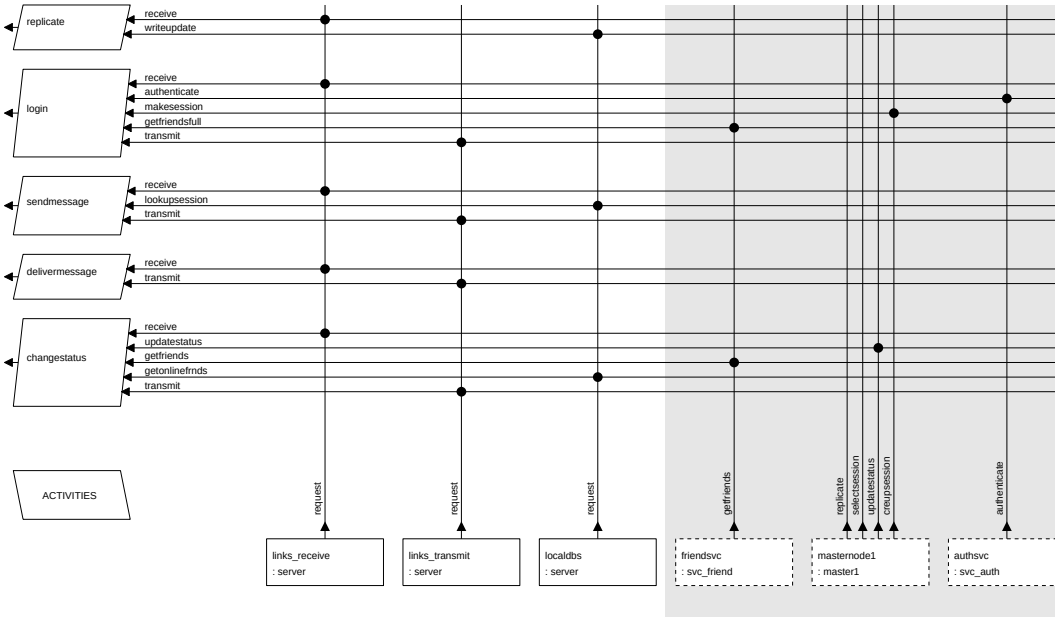


Figure 4.6: Model of slave nodes in current architecture

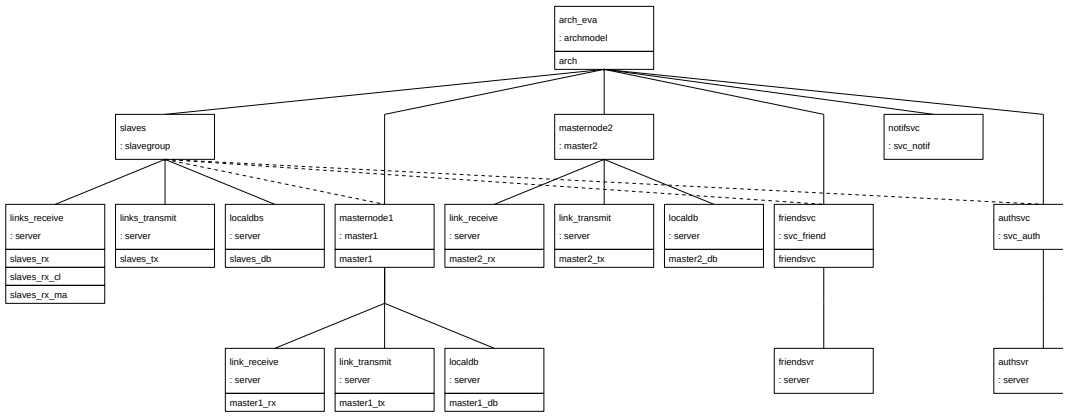


Figure 4.7: Overview of the model of current architecture

Workload parameters	
loginrate	Rate of login actions (actions per second)
logoutrate	Rate of logout actions (actions per second)
sendmessagerate	Rate of message sending (actions per second)
notificationrate	Rate of notifications (actions per second)
statuschangerate	Rate of status changes (actions per second)
numslaves	Number of slave nodes
Architecture parameters	
avgrostersize	Average number of persons in a contact list
onlinepct	Percentage of users online at peak (bytes)
avgmessagesize	Average size of an instant message (bytes)
avgstatusc2ssize	Average size of a status message, client to server (bytes)
avgstatuss2csize	Average size of a status message, server to client (bytes)
avgquerysize	Average size of a replicated write query (bytes)
avgloginc2ssize	Average size of login conversation, client to server (bytes)
avglogins2csize	Average size of login conversation, server to client (bytes)
readqueryload	Relative load of a read query on the database (workload units)
writequeryload	Relative load of a write query on the database (workload units)
linkspeed	Speed of an unidirectional network link (bits per second)
dbspeed	Speed of a database (workload units per second)

Table 4.2: Model parameters for the current architecture

4.5.2 Parametrisation

Using measurements on the current architecture, we have found values for the model parameters given in Table 4.2. These values are listed in Table 4.3. We describe how we found these values in this section.

For the average contact list (roster) size some groups of users not taken into account, as they would disrupt the statistics. Famous members, such as artists and politicians, have a lot of contacts and the chat functionality is disabled for them. Users that have not logged in for more than a week are ignored as well, as they are not representative for the average Hyves user. Of all Hyves members, four million logged in at least once during the week of the measurements. Together, they have 557 million relationships with other users, resulting in an average contact list size of 140.

Workload parameters		Architecture parameters	
loginrate	210	avgrostersize	140
logoutrate	200	onlinepct	0.06
sendmessagerate	200	avgmessagesize	200
notificationrate	240	avgstatusc2ssize	150
statuschangerate	10	avgstatuss2csize	230
numslaves	35	avgquerysize	450
		avgloginc2ssize	1150
		avglogins2csize	13,000
		readqueryload	28
		writequeryload	22
		linkspeed	1,073,741,824
		dbspeed	100,000

Table 4.3: Actual parameters for current architecture

The workload parameters were found by analysing peak load statistics and taking the 95% percentile of these statistics to remove measurement errors. The same statistics show that the maximum number of simultaneously connected users was 240,000. Combined with the number of active users, the peak online percentage of users is 6%.

To determine the load on the network, traces were captured on running slave nodes. These traces included headers all layers of the protocol stack, in this case Ethernet, TCP and IP. Analysis of the traffic between a slave node and a master node showed that the size of a packet containing a replicated update statement is about 450 bytes. The average size of a packet containing a user to user message is 200 bytes, both for the packet from the user to the slave node as well as from the slave node to the user. A notification packet is found to be around 580 bytes. Although the protocol overhead is roughly the same for a user to user message and a notification packet, the average size of the data is much larger for notification packets. The size of a packet containing a presence update sent from the client to the server is 150 bytes. However, a similar packet sent in the opposite direction is slightly larger, 230 bytes, because it contains more addressing information.

The login procedure involves a lot of messages between the application node and the client that we aggregate into two parameters. The procedure takes 1150 bytes from

client to application node and 13,000 bytes in the opposite direction. Transmission of the contact list constitutes most of the returning data. The transmission of the presence information is not taken into account, as this is modelled similar to a presence update and uses those parameters.

Database performance was measured by executing read and write statements on a database similar to one of the working system. 10,000 write statements were executed and finished processing in 2.2 seconds, yielding a rate of 4500 queries per second and 0.22 milliseconds per query. Read statements take slightly longer: 10,000 statements are processed in 2.8 seconds. The base service rate of the database server is set to 100,000 workload units per second and the workload of read and write statements are set to 28 and 22, respectively.

Two factors cause the read performance to be lower than the write performance. First, the tests are performed using the Python programming language and the results of read statements have to be converted into Python data types. The current Hyves IM&P service uses Python in the same way, so this behaviour is comparable. The second reason relates to the storage of the database. The entire database is kept in main memory, which has a much higher write performance than hard disks. While writing to main memory is still slower than reading from it, both operations are multitudes slower on hard disks.

The speed of all links in the system is 1 gigabit per second, which is 1,073,741,824 bits per second.

4.5.3 Analysis

In this phase, the abstract analytical model is used analyse the current architecture. First, we analyse the performance limits of the current architecture using the found parameters. After that we analyse the scalability issues discovered in Section 3.7.

Using performance analysis we determine the capacity limit of the current architecture. This limit is reached when the utilisation of one of the components of the architecture reaches 100%.

The parameters found in Section 4.5.2 were entered in the current model. The model was analysed using one hundred different workloads, ranging uniformly from 10% of the current peak workload values to 1000% of these values. At the same time, the

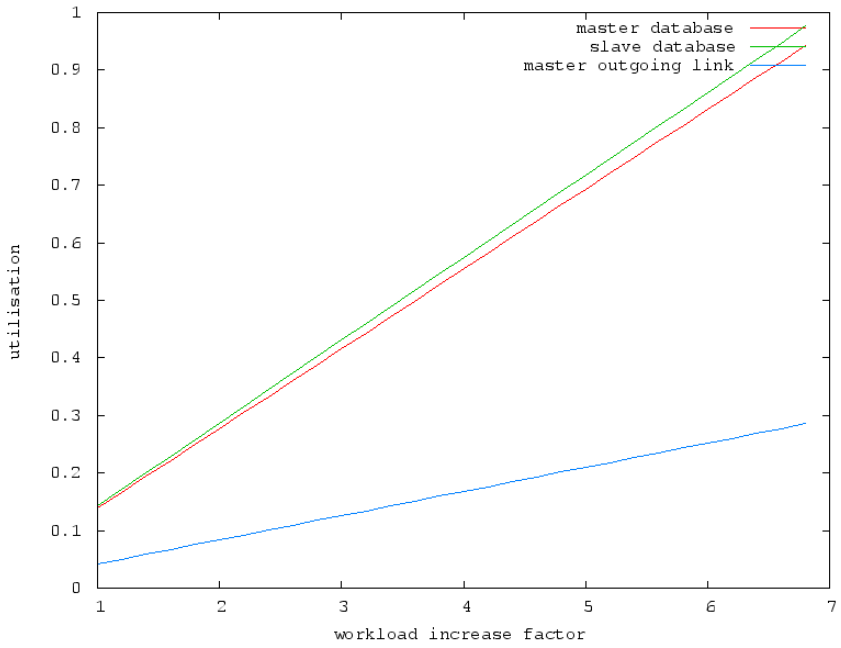
number of nodes is kept constant at the current number of nodes. The peak workload parameters are given in Table 4.3.

Figure 4.8(a) shows the utilisation of one slave database and the database and the outgoing link of the first master node for increasing workloads. The x-axis contains the factor by which the workload is multiplied. The y-axis contains the utilisation for each component, where zero means idle and one means fully utilised. The linear curve indicates that the increase in utilisation of each component analysed is proportional to the increase in workload. It shows that the slave database reaches 100% utilisation the first, with the master database following closely. The load on the slave database is slightly higher, as it is accessed by the application node for read queries and by one of the master nodes to replicate write queries. According to HIT, at more than 6.8 times the current peak workload the database is the first to reach its maximum capacity as its utilisation exceeds 1. At the same time, the utilisation of the first masters' outgoing link is just 0.3, or 300 megabit per second.

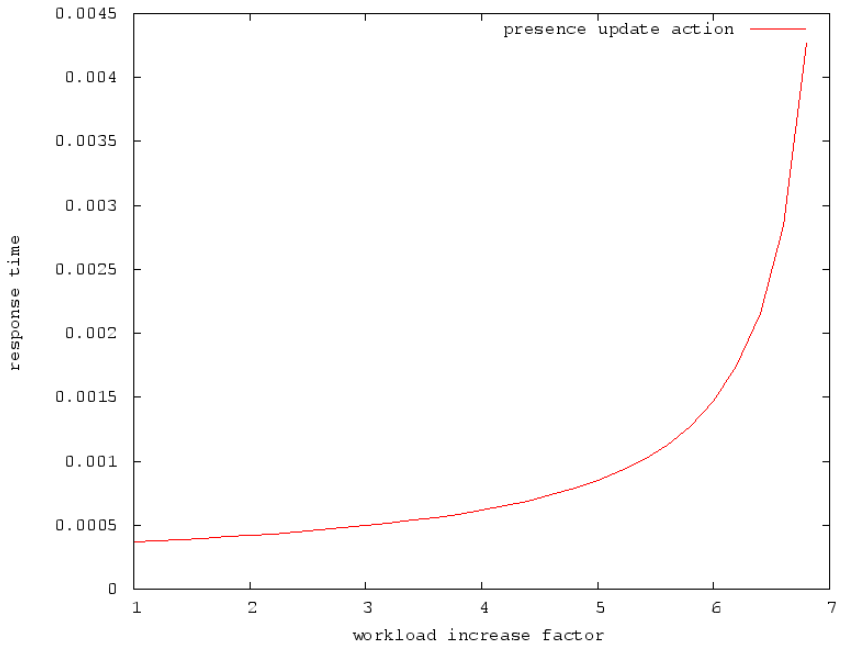
Figure 4.8(b) shows the response time of the presence update action for the same range of workloads. In this graph, the y-axis shows the response time of the model in seconds. This is not the entire response time for this request, as the model contains only a portion of the actual process. It is however indicative of the behaviour of the model to a high load. The exponential curve indicates that the response time becomes infinitely high when the workload approaches seven times the current workload. The shape of the curve relates to the fact that for an increasingly loaded database or network link, new queries or jobs have to increasingly wait longer to be served. Between six and seven times the current workload the response time is already increasing rapidly. Based on this result, the maximum workload that can be handled by this architecture is around six times the current workload.

The bottlenecks described in Section 3.7 relate to the write load on the first database master, the outgoing traffic of the first database master and the write load on each database slave. In the following experiment we increase the workload from the current workload to a ten times multiple of it. At the same time, we increase the number of slaves by the same rate. If the architecture were scalable, adding more nodes would allow handling a larger workload.

Figure 4.9 shows the results of this experiment on the master and slave databases. The x-axis contains the factor by which the workload is multiplied, which is equal to



(a) Utilisation of databases and links



(b) Response time of presence update

Figure 4.8: Analysis of proportionally increasing loads on the current architecture

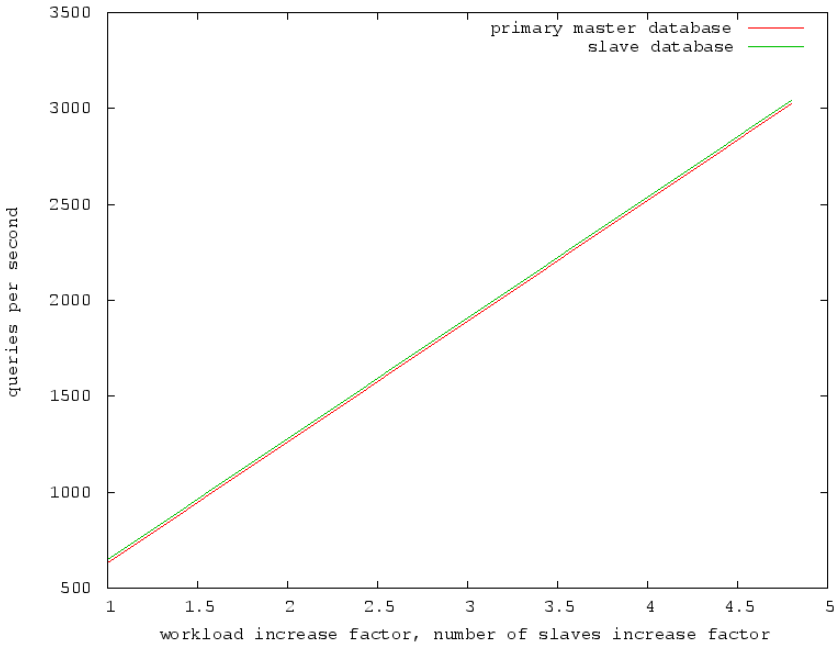
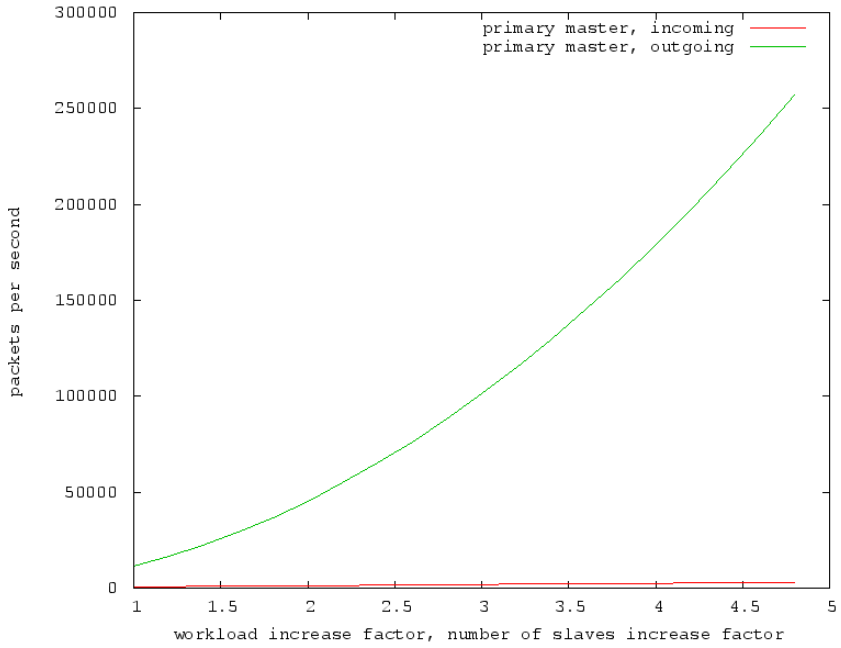


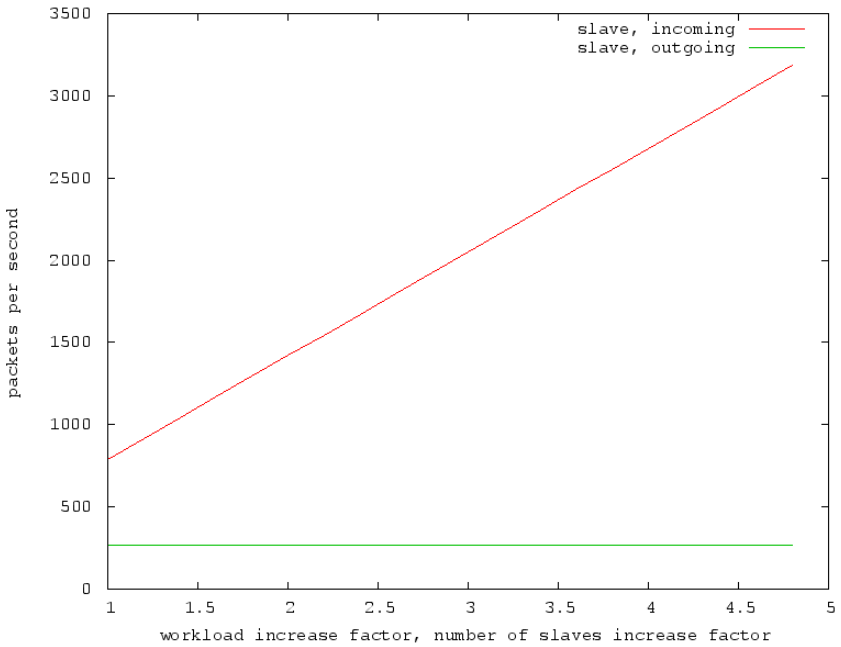
Figure 4.9: Throughput of first master and slave databases

the factor by which the number of slaves is multiplied. The starting values are those listed in Table 4.3. The y-axis shows the number of queries per second for the first master database and one slave database. Two observations can be made: first, both numbers increase linearly with the increased load, despite the fact that more slaves have been added. Secondly, the two curves are quite close to each other, with the slave database slightly more loaded than the master database. This relates to the fact that the slave database handles both the replicated update queries arriving from the master and the read queries executed by the application. The maximum workload in this experiment is 4.8 times the current workload, which is less than the 6.8 of the previous experiment. The difference is that now we add additional slave nodes, which causes the master databases to become overloaded even sooner than before.

Figure 4.10(a) shows the effect of the same experiment on the incoming and outgoing network traffic of the first master node. Here, the y-axis contains the number of packets going in and out of the master node. The curve of the outgoing link shows the quadratic behaviour of this traffic, as number of outgoing packets is influenced by the increasing number of slaves and the increasing workload. The incoming traffic related to only the



(a) Throughput of master network link



(b) Throughput of slave network link

Figure 4.10: Scalability analysis of network traffic in the current architecture

latter of those factors.

Figure 4.10(b) shows the results of this experiment on the network link of a slave node. The x-axis and y-axis have the same properties as previously. The linear increase in incoming traffic is related to the incoming replication updates. The outgoing traffic remains constant, as this is the traffic to the clients. As more slaves are added proportionally to the increased workload, each slave handles the same number of requests from clients throughout the experiment.

The analysis shows that all resources related to the replication of presence updates are contributing to the non-scalability of this architecture. The network links and the databases of both the database master node and the slave nodes overload when the workload is increased. Adding slaves only aggravates the problems at the master node.

5

Architecture Proposals & Analysis

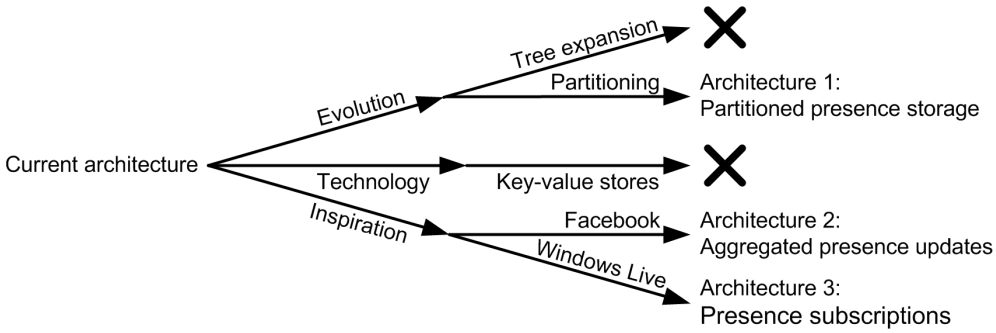


Figure 5.1: Overview of the considered approaches

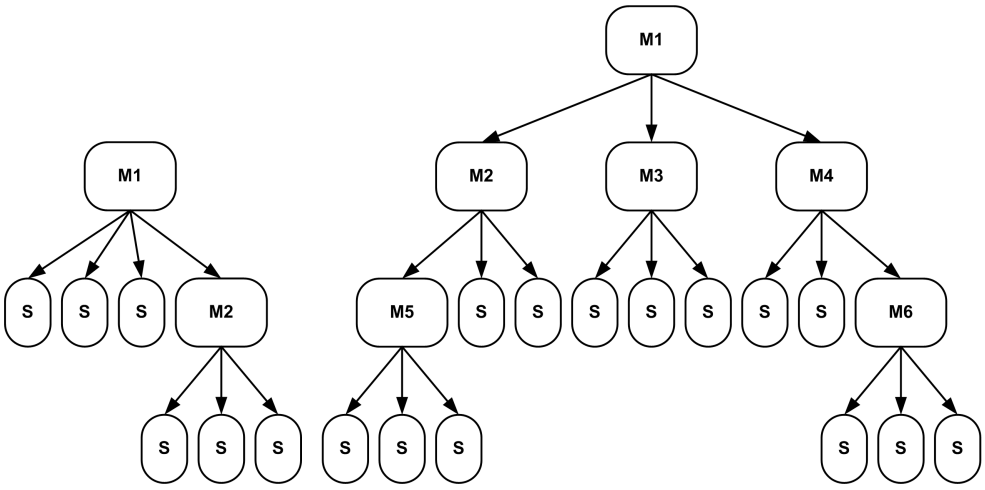
Figure 5.1 gives an overview of the architectural approaches described in this chapter, including two that do not lead to a proposal. The approaches are categorised in three directions: evolution from the current architecture, adding new technology and using elements from other architectures.

Evolving the current architecture can be done both by extending the replication tree and by partitioning the database nodes. Extending the tree does not remove the bottlenecks, as we will describe in Section 5.1. A partitioned architecture is discussed in Section 5.2. A technological advancement that is investigated for other services of Hyves is the key-value store. We discuss in Section 5.1 why this approach does not give architectural benefits. We analyse two new architectures using information

from the instant messaging & presence architectures of Facebook and Windows Live Messenger in sections 5.3 and 5.4. Facebook uses aggregation to reduce the amount of traffic caused by presence updates, while the Windows Live Messenger architecture uses subscriptions to distribute presence information.

5.1 Discarded approaches

In the current architecture, the hierarchy of database master and slave nodes resembles a tree, as is shown in Figure 5.2(a). The first master is the root, the slaves are the leaves of the tree and the second master is an intermediate node. To decrease the load of the outgoing link of the first master node, new intermediate nodes can be added, which reduces the number of slaves attached directly to the first master node. A possible result of this approach is given in Figure 5.2(b). Unfortunately, this solves only a part of the problem. The load on the incoming network links of the database nodes and the processing load do not decrease.



(a) Current architecture (b) Example of extended tree

Figure 5.2: Database node hierarchies

Another approach that was investigated but abandoned involves key-value stores. Key-value stores are databases with much less features than relational database management systems (RDBMSs). In essence, they only allow looking up an unstructured piece of data (*value*) by some identifier (*key*). Features such as sorting, selection of

attributes and grouping are not available. The advantage is that these stores are better suited for distribution, as exactly those features make distribution difficult. Due to their simplicity, they are easier to manage and have a better performance than traditional RDBMSs.

The key-value paradigm does suit the presence data model. An identifier for each user, such as the unique username or the numeric user id, could be used as a key, and all presence information could be used as the value. The data access patterns in the current architecture are supported by key-value stores.

Even though key-value stores can change the performance of the presence storage, the nature of an architecture as a whole is not influenced. The function of a key-value store in an architecture would not differ from that of an RDBMS.

5.2 Architecture 1: Evolutionary partitioning

In the following we discuss architecture proposals that have the potential to elevate scalability using the same building blocks as the current architecture. By partitioning the master node and slave nodes (see Section 2.4) increased load can be handled.

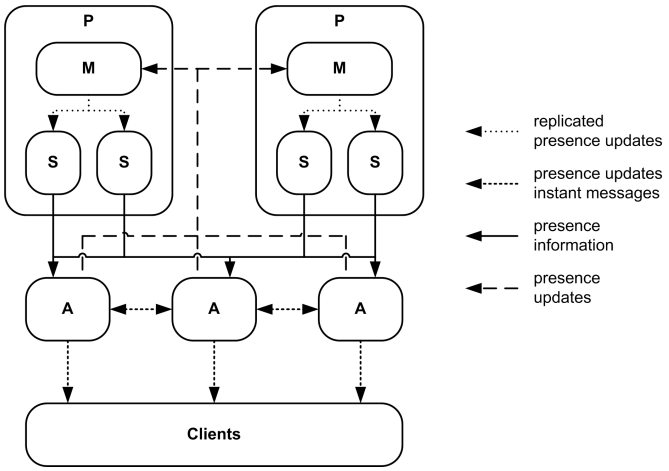


Figure 5.3: Overview of the evolutionary improved architecture

5.2.1 Overview

Figure 5.3 shows the new composition of the components in this architecture. The architecture contains *application nodes* that have the same role as in the current architecture. However, in this architecture the database *slave nodes* are separate machines that are grouped in one or more *partitions*. Each partition consists of one database *master* and one or more database slave nodes. Each user is assigned to a partition by a partitioning algorithm such as Consistent Hashing (see Section 2.4). This means that the presence information is divided over many partitions. When a user logs in, the application node needs to look up presence information of the users in the contact list on all partitions. Using asynchronous programming, queries to all partition can be executed in parallel, reducing latency.

Users can connect to any application node available. Upon reception of a presence update from a client, it submits the update to the master node of the users' partition, which replicates it to the slaves of that partition. Instant messages are transmitted directly between application nodes, similar to the current architecture.

Application nodes are not associated with just one partition because this would requires the client to contain the partitioning scheme to select the right application node. This could be circumvented by installing a load balancer to redirect new connections to the right set of application nodes, but this creates extra architectural complexity.

Furthermore, balancing the load over different application nodes is easier when they are all the same. When an application node is fixed to one partition, a precise measurement of the load on that partition is required to determine the number of application nodes. To determine the right number of application nodes in the proposed situation, only the average load on all application nodes needs to be determined.

Finally, when each application node is associated with just one partition, one or two application node failures can quickly cause one partition to become unavailable, as there are just a few application nodes per partition. With equal application nodes one or two failures are not fatal, as long as the other application nodes have spare capacity.

5.2.2 Scalibility and remaining bottlenecks

The capacity of this architecture can be increased in three ways, depending on the kind of capacity needed. Increasing the number of application nodes increases the connection

capacity of the architecture. More application nodes also have to be added when new features are introduced that require more processing power at the application layer. Increasing the number of partitions is necessary when the number of write queries on the databases increases. This can either be caused by an increased number of users or an increased level of activity. The key action that increases the number of writes is the presence update, which is also included in the login and logout action.

Within each partition, slave nodes can be added to cope with increased read load. Read queries are executed for all actions on the system. For instance in the case of sending a message, the application node has to check if the recipient is available.

In practice, a combination of the options above has to be applied when larger capacity is needed. Increased load requires more application nodes as processing is required for each action. Increased presence updates require more partitions, but at the same time the number of slaves within each existing partition must also be increased.

While it might seem that this architecture solves all bottlenecks of the current architecture, some limitations to this architecture might appear in the long run.

Network While the network layout and geographical distribution is not much of an issue with the current architecture, this proposed architecture has more strict network requirements. The response time of the system greatly depends on the read speed of presence information in the database. In the current architecture, the read speed is high because the database slave node is located on the same machine as the application node. Even when all partitions are accessed in parallel by the application node, a read query becomes slower with the proposed architecture. It is important that the application nodes are close to the partition and the components of the partition are close to each other in terms of network topology.

Slave explosion In theory, increased load can be answered by adding a new partition, providing linear capacity growth. However, the read load on all existing partitions is increased as well, requiring more database slaves at those partitions.

5.2.3 HIT model

Just as the current architecture, this architecture is modelled using HIT. In contrast to the model of the current architecture, application nodes and database nodes are

modelled separately as they are split over multiple machines.

The top layer, displayed in Figure 5.4, forwards action invocations to services of the application nodes. The application nodes are modelled by the *applnodes* component, in Figure 5.5. This model shows the incoming and outgoing links of the application nodes as well as the *partitions* component, representing all partitions.

The application nodes forward requests to partitions, which consist of a master and several slaves. Figure 5.6 shows how read queries are forwarded to a slave and write queries to the master node. The *replicate* service is used by the asynchronous replication process.

Table 5.1 contains new parameters used by this model, in addition to the parameters of the current architecture. The *avgpresencesize* parameter contains the number of bytes of presence information stored per user. In the current architecture, this information is transferred locally between the application node and slave node. In this proposal these nodes are located on separate machines, raising the need to model the size of this data. The amount of presence information currently stored in the database per user is about 700 bytes.

Workload parameters		
numap	Number of application nodes	
numpar	Number of partitions	
numspp	Number of slaves per partition	
Architecture parameters		
avgpresencesize	Size of presence information per user (bytes)	700

Table 5.1: Additional model parameters for architecture proposal 1

5.2.4 Analysis

We use the HIT model to determine if this architecture scales where the original architecture did not scale. In Section 4.5.3, we discovered that the throughput of the links and databases of each component in the current architecture increases regardless of the addition of nodes. In the following experiments we investigate for each type of node if this proposal exhibits this issue. For each experiment the workload factor is varied

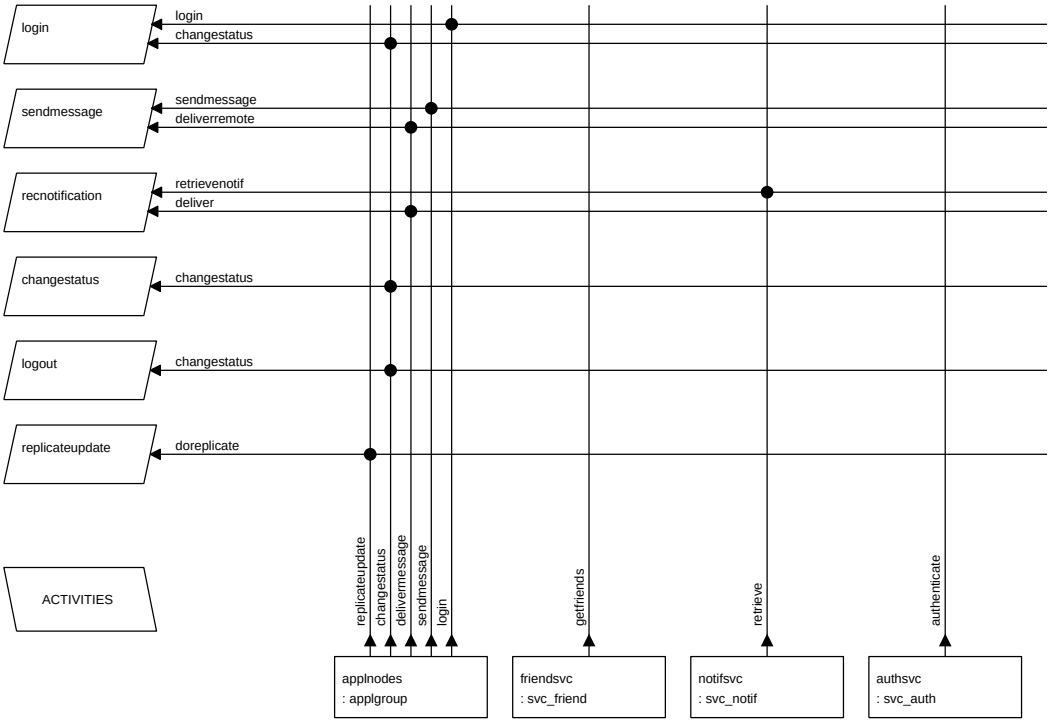


Figure 5.4: Top layer of model of architecture proposal 1

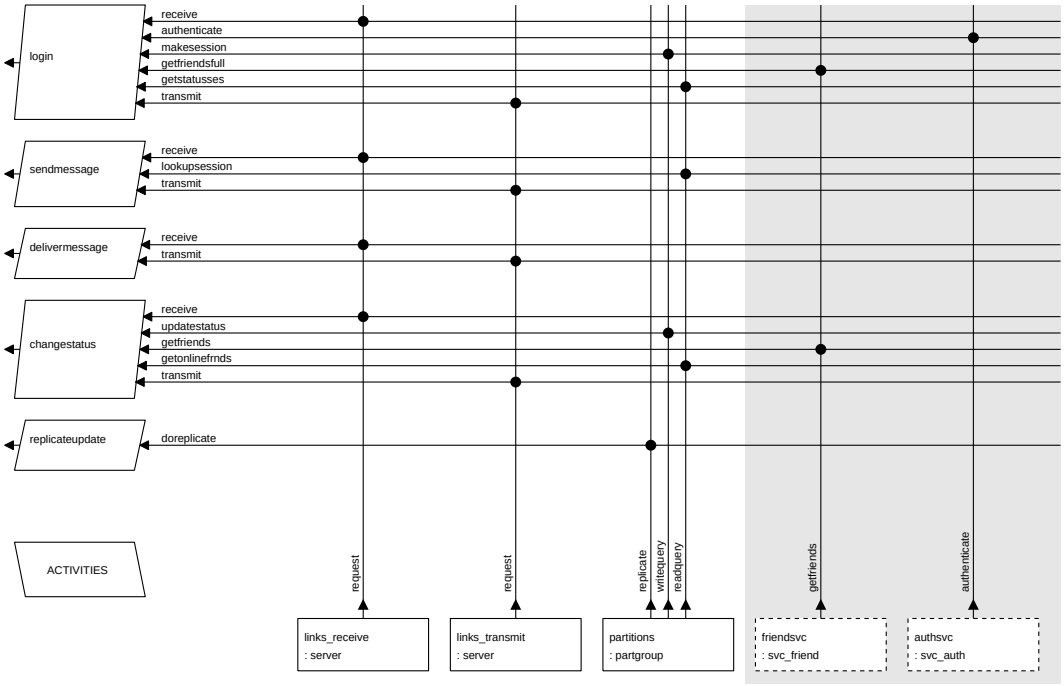


Figure 5.5: Model of application nodes in architecture proposal 1

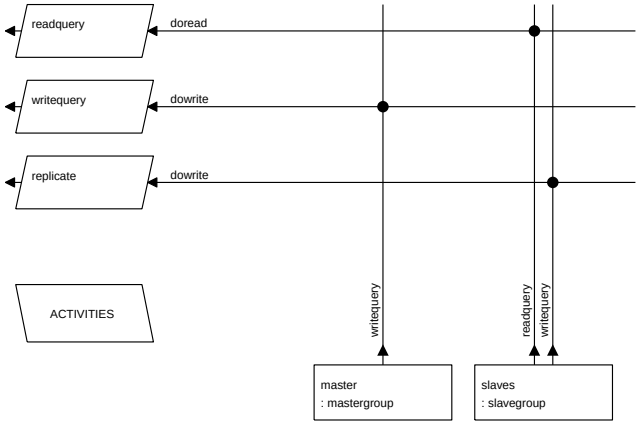
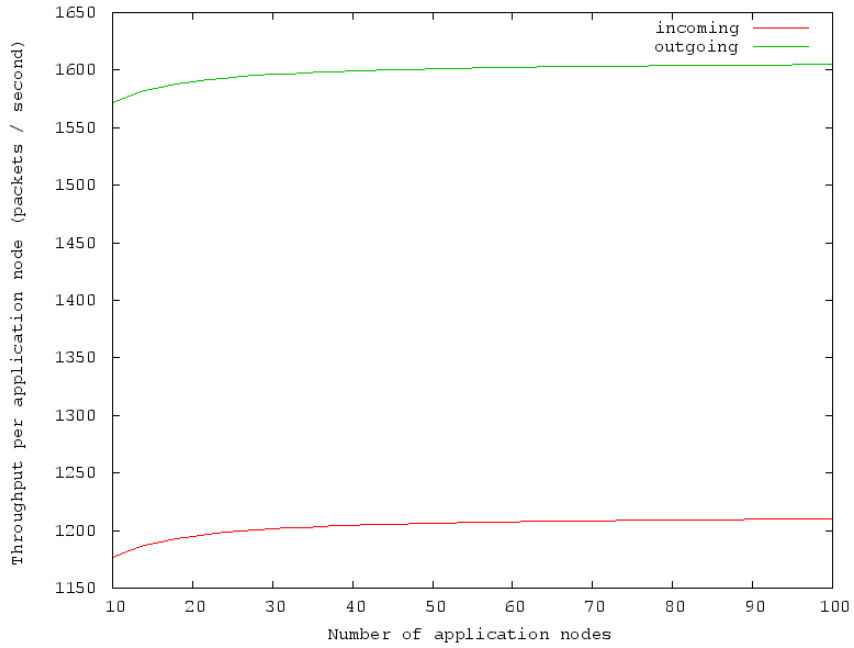
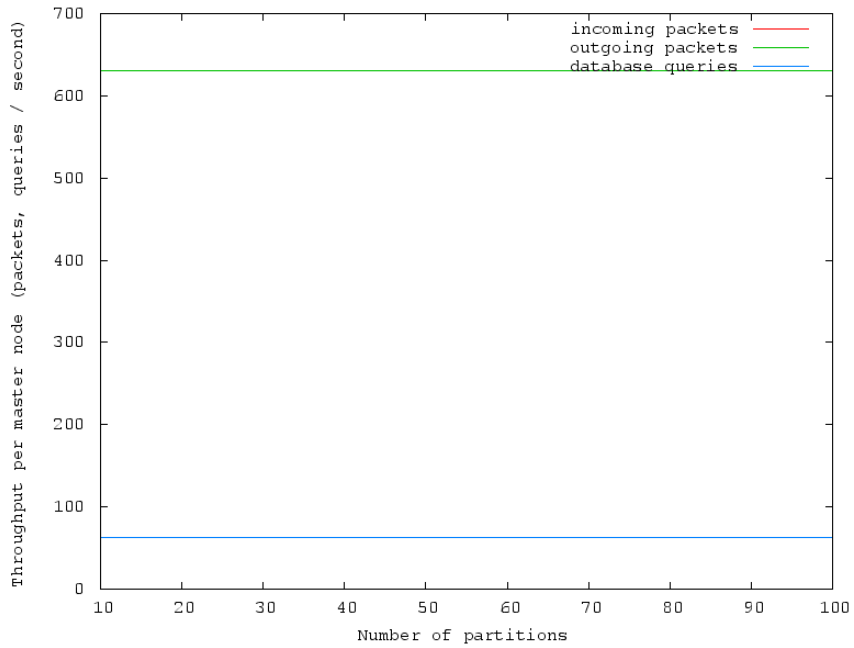


Figure 5.6: Model of partitions in architecture proposal 1

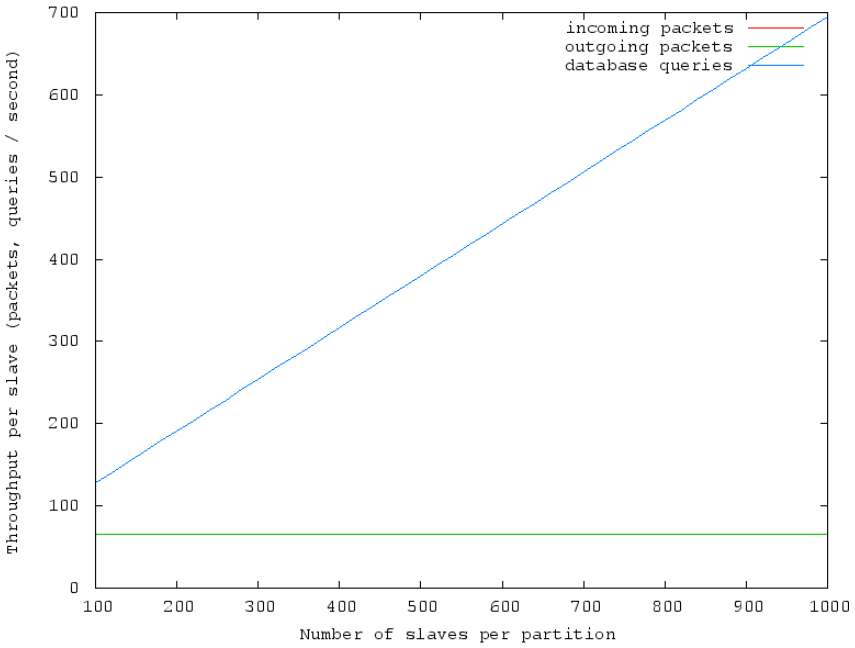


(a) Throughput of application nodes

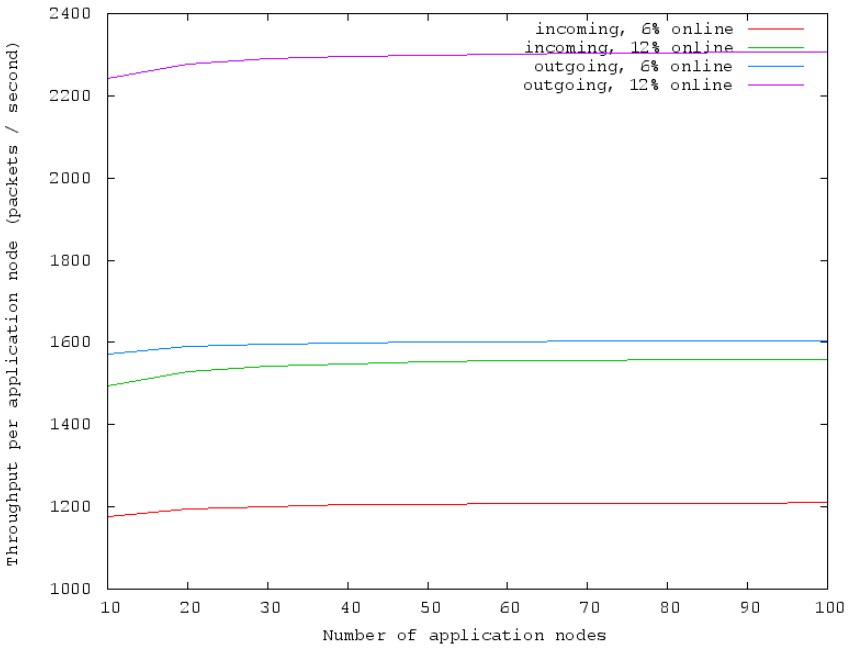


(b) Throughput of master nodes

Figure 5.7: Analysis of proportionally increasing loads in architecture proposal 1



(a) Throughput of slave nodes



(b) Throughput of application nodes for different parameters

Figure 5.8: Analysis of proportionally increasing loads in architecture proposal 1

from one to ten, while each parameter of the peak workload rates is multiplied by this factor.

Figure 5.7(a) shows the throughput at the incoming and outgoing links of the application node. The x-axis displays the number of application nodes. We have chosen to increase the number of application nodes by ten for every increase of the workload factor. The exact value of this number does not matter for our analysis, as this is only important for performance, not for scalability. The number of partitions and the number of slaves per partition have been kept constant at ten during this experiment. The number of packets sent and received per application node is given on the y-axis. The curves start nonlinear, but eventually converge to a constant value. This means that the load per application node is constant for a proportional amount of application nodes per unit of workload, indicating good scalability.

Figure 5.7(b) displays the results for the second experiment. Here, the number of partitions is increased by ten for every increase in the workload factor. The curves of the number of incoming packets and the number of queries are equal, indicating that every incoming packet results in a query. The curve of the number of outgoing packets is exactly ten times the number of incoming packets because each query is replicated to ten slaves.

Finally, Figure 5.8(a) gives the throughput of the resources of the slave nodes. With the number of application nodes and the number of partitions held constant at ten, the number of slaves per partition is varied from ten to one hundred. As a result, the total number of slaves varies from one hundred to one thousand. The number of incoming packets is equal to the number of queries, which increases linearly with the workload. This shows that by itself, the increasing the number of slaves per partition does not solve scalability problems. The number of partitions has to be increased as well to limit the load per slave node.

As explained in Section 4.4, we also change other parameters than just the workload. Figure 5.8(b) gives the results of the same experiment as Figure 5.7(a), but now for two different values of the *onlinepct* parameter. The original peak percentage of online users is 6%. Doubling this value causes a slight increase in the load on the network links of the application nodes. For other node types, changing this parameter had no influence.

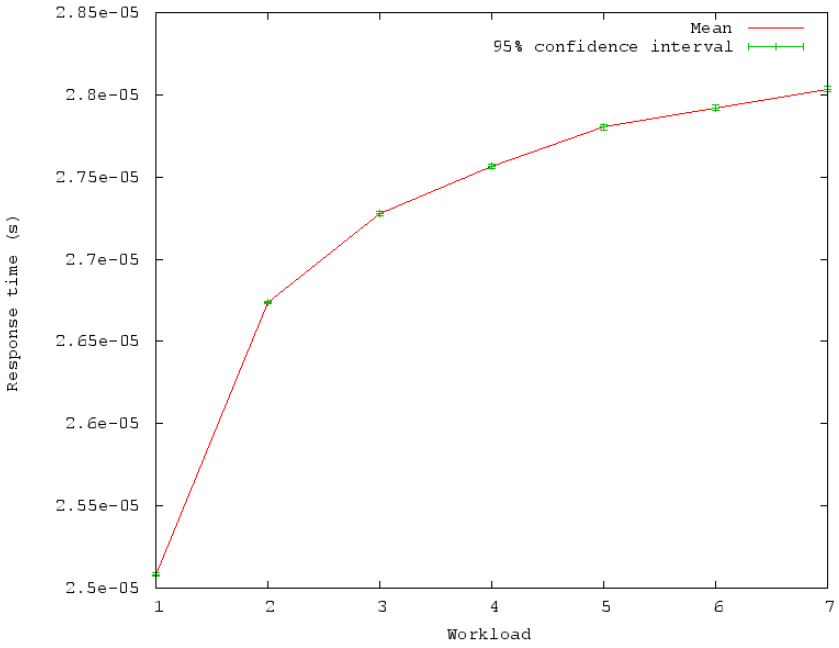


Figure 5.9: Response time of message sending in architecture 1

The response time of the message sending action is given in Figure 5.9. The workload and number of application nodes, partitions and slaves per partition is varied from 1 to 7, as higher numbers were unsolvable using the HIT simulator. The graph shows that the response time approaches a constant value after an initial increase. This is caused by the increasing probability that the sender is connected to a different application node than the recipient. The response time of sending an instant message is thus limited in this architecture. However, as the number of slaves per partition as well as the number of partitions increases, the total number of slaves increases quadratically.

We can conclude that the application and database master nodes of this architecture are linearly scalable. However, scaling the database slave nodes requires adding both slaves to partitions and new partitions. This results in a quadratic increase in number of machines. Hence, this architecture does not scale linearly.

5.3 Architecture 2: Aggregated, batched presence updates

The Facebook chat architecture provides Instant Messaging & Presence to nearly 200 million users (see Section 2.6.2). Instead of distributing presence updates instantly to other users, this architecture batches presence updates and sends them at fixed intervals. They use the partitioning approach just for exchanging messages and keep all presence information on centralised presence nodes. We use this approach to define an architecture that is suitable for Hyves.

5.3.1 Overview

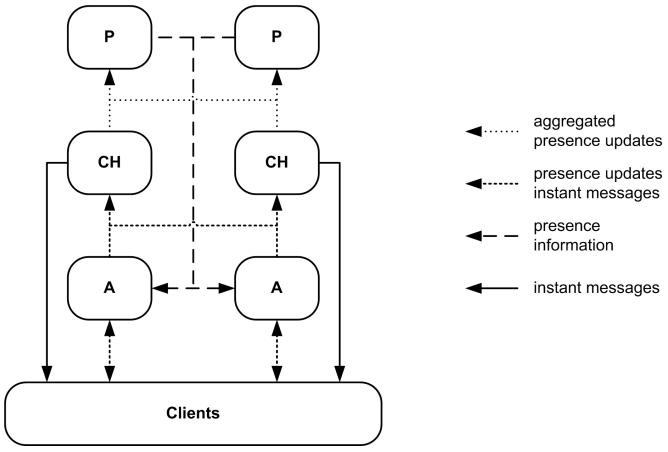


Figure 5.10: Overview of the aggregated and batched presences approach

Figure 5.10 gives an overview of this architecture. Each channel node forwards messages and stores presence information for a portion of the user base. Clients connect to one of the *application nodes* and the *channel node* that stores their messages. Application nodes forward incoming messages to the channel node of the receiving user. Incoming presence updates are forwarded to the channel node of the user that sent the presence update. They are stored at channel nodes and transmitted in batches at a given interval to all *presence nodes*, which store presence information for all users. Application nodes poll presence information for their connected users at a given interval. They fetch presence information for all users from the contact lists and forward this information to their clients.

Channel nodes are the authoritative source for presence information, while application nodes do not store presence information at all. When an application node tries to send a message to a channel node for a user that is unavailable, the channel node generates an error, which is transported back to the client. The capacity requirements on their links force us to reduce the amount of presence information kept per user. At this moment, a user can select one of six possible presence statusses as well as store information in a 500 byte free text field. Only the presence status can be stored in this architecture. We store this information using three bits of data.

5.3.2 Scalability and remaining bottlenecks

With increasing load, new application nodes can be added easily. Adding new channel nodes to cope with an increasing number of users is easy as well, as new channel servers can serve new users and no migration has to take place. To handle more active users, more channel nodes can be added to relieve the existing channel nodes. A part of the users has to be migrated to these new channel nodes, potentially causing disruption of the service. New presence nodes can be added when the read load increases. It is not possible to add more presence nodes when either the write load grows or the size of the presence information becomes larger than the size of the main memory at each node.

The advantage of the presence update approach of this architecture is that the load on the system is no longer determined by the number of presence updates, logins and logouts, but by the number of registered and online users. The number of online users determines how much data is fetched from the presence nodes by the application nodes. The number of registered users determines the traffic between the channel servers and the presence servers. Both numbers indirectly determine the required number of presence servers.

In this architecture, the presence nodes form a theoretical bottleneck in a couple of ways. The maximum number of users that the architecture can support depends on the amount of main memory that is available. We have been able to reduce the amount of data per user to three bits. Storing presence information for all registered users would require just three megabytes, a negligible amount of memory. Storing the original amount of one kilobyte of data per user is even feasible with current memory prices, but this would lead to congestion of the network links.

The second bottleneck posed by the presence nodes is their write capacity. Similar to the current architecture, each presence node receives all presence updates, although the approach of batching updates might decrease the load per update. However, the write capacity of the presence nodes limits the scalability of this architecture.

The third bottleneck is the load on the channel nodes for sending the updates to the presence servers. When the number of presence nodes grows to cope with the read load, channel nodes need to submit their updates to an increasing number of presence servers, potentially overloading their outgoing links.

5.3.3 HIT model

Figure 5.11 contains the top layer of the HIT model for the second proposal. Important differences between this proposal and the previous proposal are the three asynchronous processes *retrpresence*, *pushpresence* and *propagpresence*. *retrpresence* invokes the *appnodes* component to retrieve presence information from presence nodes. The *pushpresence* service invokes the *channodes* component to push presence information from channel nodes to presence nodes. The *propagmessage* service transfers messages from channel nodes to clients. The model of the application nodes is given in Figure 5.12. It shows that application nodes communicate with clients, presence nodes and channel nodes. The model for the channel nodes is given in 5.13. It shows the services that channel nodes offer to application nodes and clients.

Figure 5.14 gives the model for the presence nodes. It is a simple get/set model. The *savepresences* services saves presence information for a large number of users and is invoked by the channel nodes. The *getpresences* service retrieves presence information for one user.

Table 5.2 contains the model parameters specific for this architecture proposal. Most of the parameters of the current architecture are used by this model as well, including their values. The number of registered members and the average size of a contact list are the same as used in Section 4.5.2. The size of the presence information per user is three bits, as determined earlier. Each person is identified by a four byte integer. Taking some protocol overhead into account, 600 bytes for a request from an application node to a presence node is reasonable. The presence push rate and presence fetch rate are equal to the values used by Facebook.

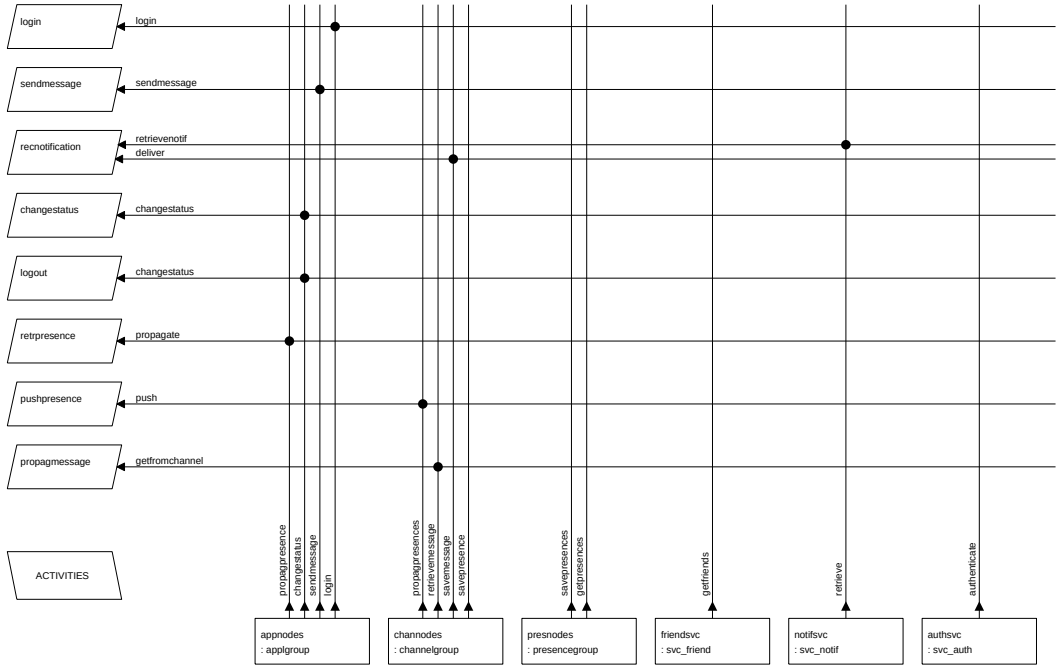


Figure 5.11: Top layer of model of architecture proposal 2

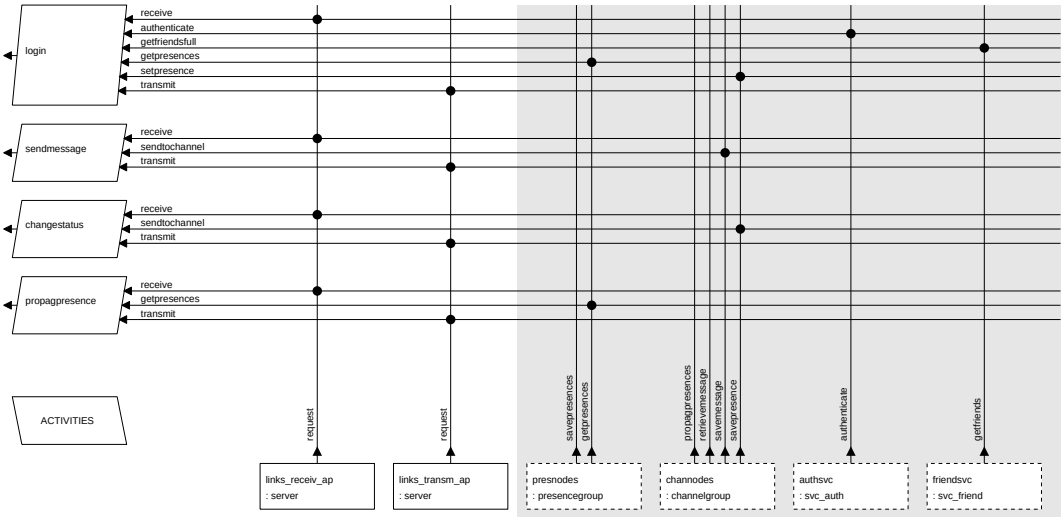


Figure 5.12: Model of application nodes in architecture proposal 2

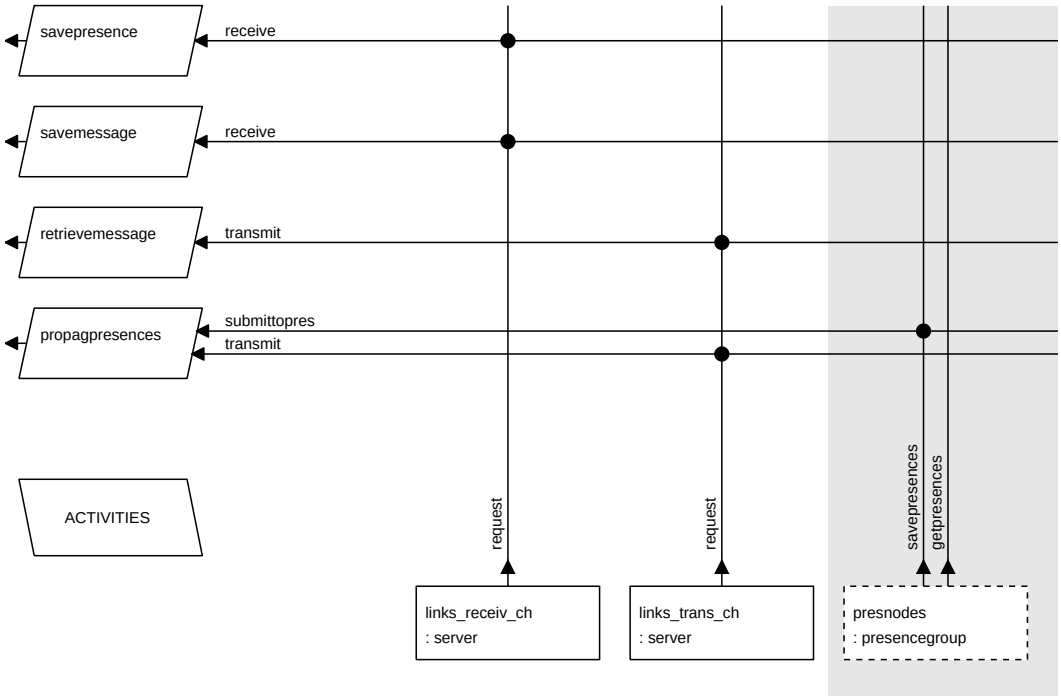


Figure 5.13: Model of channel nodes in architecture proposal 2

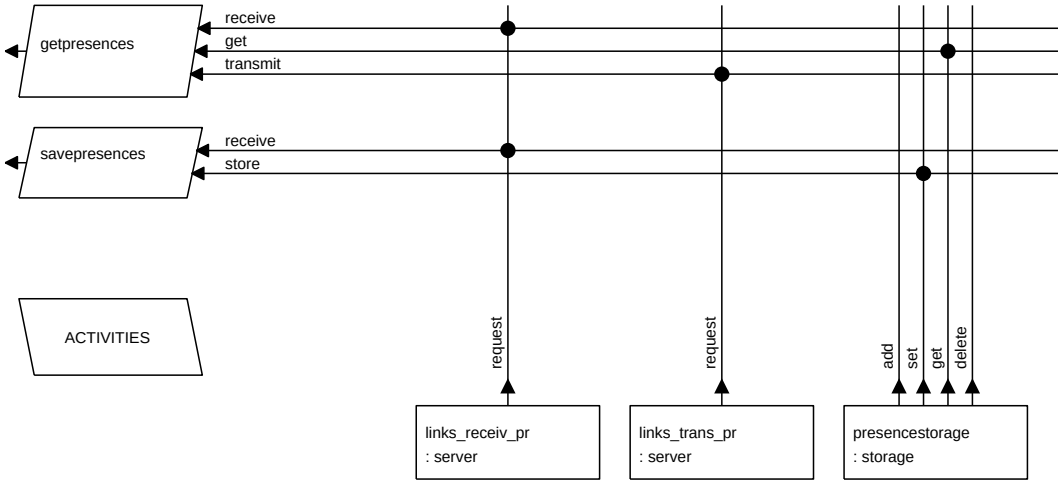


Figure 5.14: Model of presence nodes in architecture proposal 2

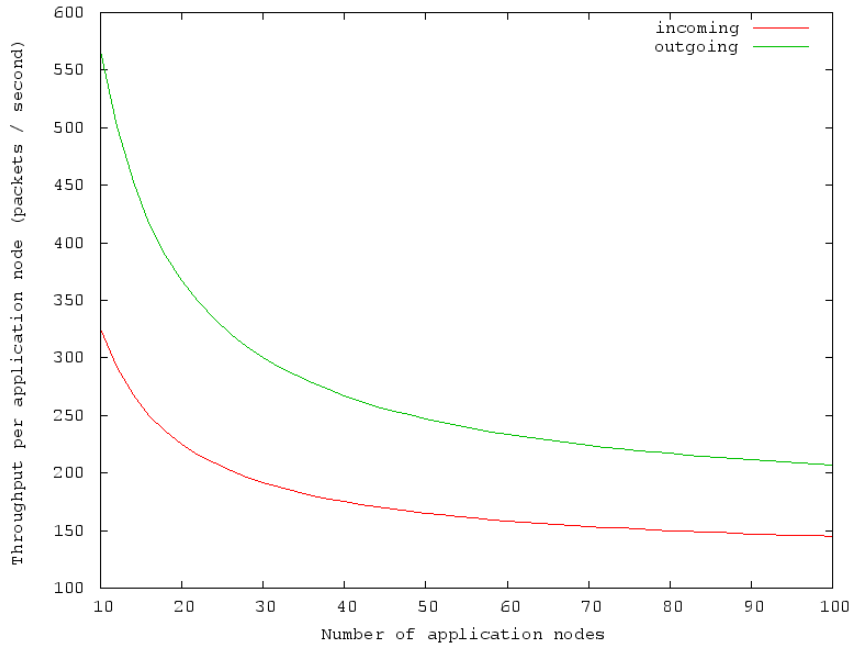
Workload parameters		
numap	Number of application nodes	
numch	Number of channel nodes	
numpr	Number of presence nodes	
Architecture parameters		
nummembers	Number of registered members	4,000,000
avgpresencesize	Size of presence information per user (bytes)	0.375
avgpresencepollsize	Average size of a request from application nodes to presence nodes (bytes)	600
presencepushrate	Rate by which presence information are pushed from channel nodes to presence nodes (seconds)	30
presencefetchrate	Rate by which presence information is fetched from presence nodes by application nodes (seconds)	120

Table 5.2: Additional model parameters for architecture proposal 2

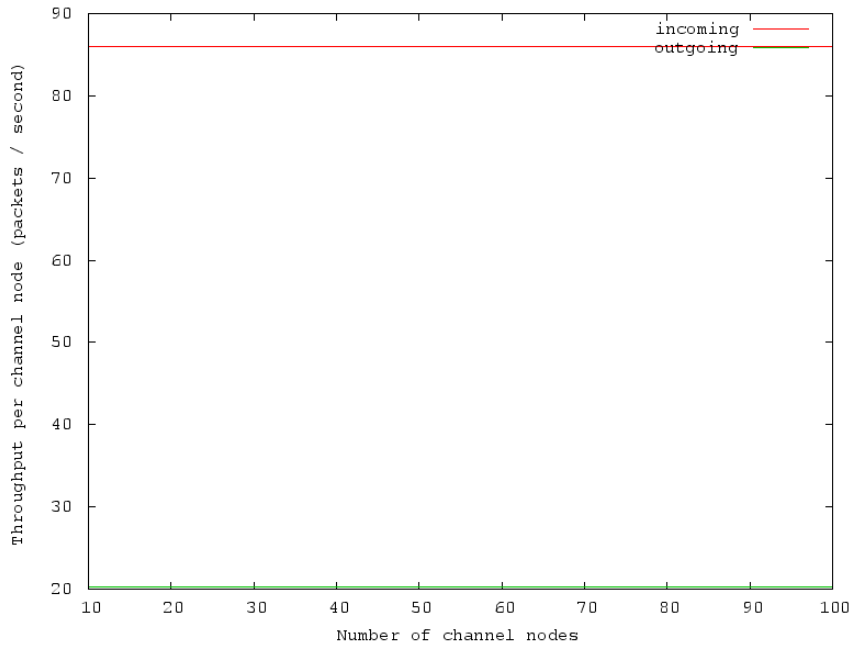
5.3.4 Analysis

We analyse the scalability of this architecture in a similar way as the previous architecture. Again, the workload and one architecture parameter are varied proportionally while all other architecture parameters are kept constant.

Figure 5.15(a) gives the throughput of the resources at each application node. The x-axis shows the number of application nodes, while y-axis shows the number of incoming and outgoing packets. The number of channel nodes and presence nodes in this experiment is ten. The workload increases from the current peak rates to ten times the current peak rates, proportionally to the increase in the number of application nodes. The curves indicate that the number of packets is nonlinear with respect to the changing workload. This is caused by the constant polling for presence updates, which is independent from the actual rate of presence updates. The load generated by these polls amortises over the number of application nodes, causing the quadratic decrease of traffic. As the throughput will eventually be linear, this component is considered to be scalable.

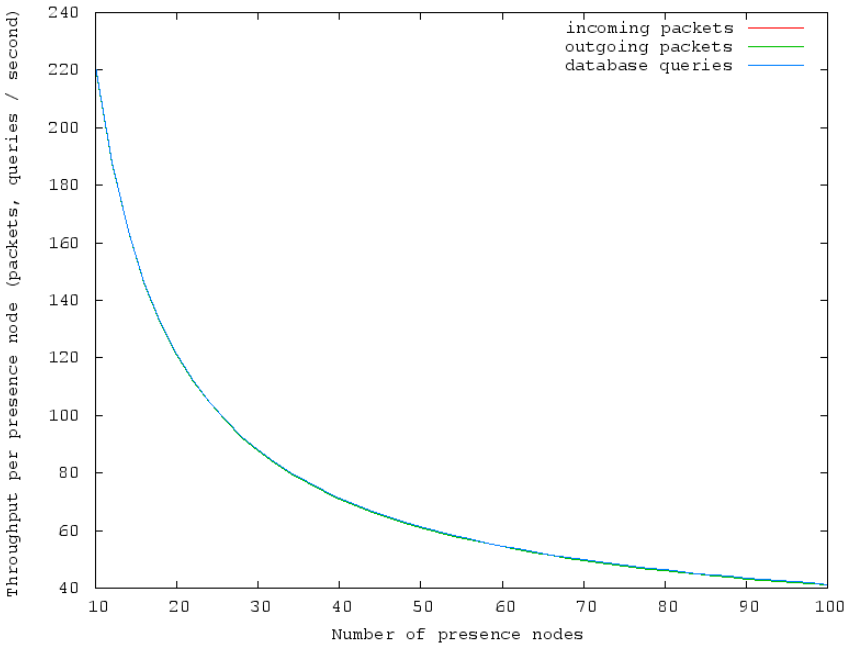


(a) Throughput of application nodes

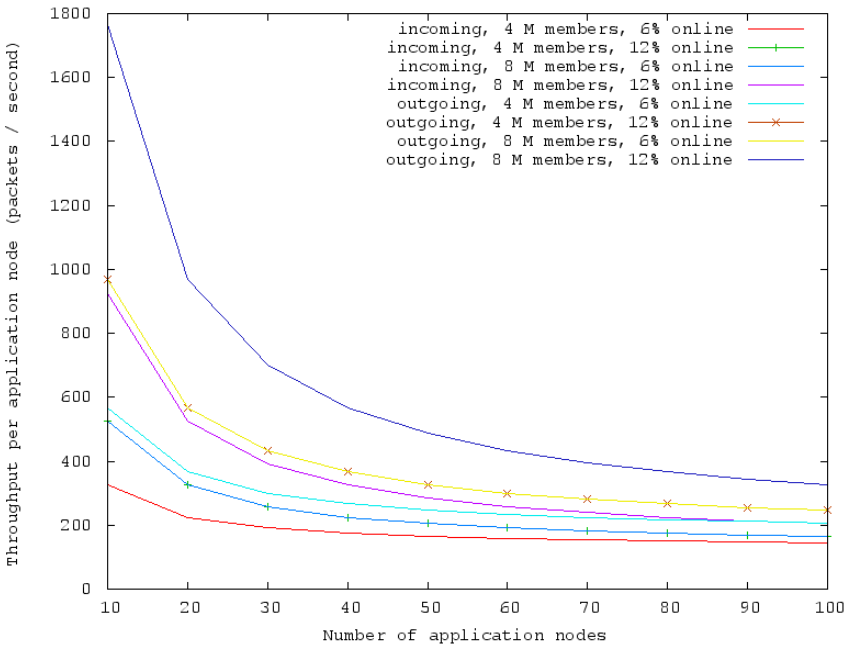


(b) Throughput of channel nodes

Figure 5.15: Analysis of proportionally increasing loads in architecture proposal 2



(a) Throughput of presence nodes



(b) Throughput of application nodes for different parameters

Figure 5.16: Analysis of proportionally increasing loads in architecture proposal 2

Figure 5.15(b) gives the throughput for the incoming and outgoing network links of a channel node. These throughputs are the result of presence updates, message sends and the constant pushing of presence updates from channel nodes to presence nodes. The constant curves indicate that this component is scalable.

Figure 5.16(a) contains the throughput of each resource in a presence node. The number of queries and the number of incoming packets are equal, as each incoming packet leads to a query to the database. The number of outgoing packets is slightly less as the model does not generate a response packet for incoming presence information. Again, these curves indicate that the load on this architecture is not dependant on the rates of the actions. While the workloads increase, the throughput of each presence node decreases.

The experiment was repeated for doubled values of *nummembers* and *onlinepct*. Figure 5.16(b) shows the results for the application node. It shows that doubling the number of members or doubling the online percentage has the same result for both incoming and outgoing traffic. Doubling both nearly quadruples the traffic. The graph shows that there is almost a direct relation between the two parameters and the throughput in the application node. Experiments on the presence nodes showed a similar result. On the other hand, the channel nodes were not influenced at all by changes in these parameters.

Figure 5.17 shows the response time of sending instant messages. The x-axis shows the number of application, presence and channel nodes and the workload. The y-axis gives the response time in seconds. The graph is constant, which means that scaling this architecture does not affect the response time of an instant message at all. An instant message always passes one application node and one channel node, regardless of the amount of nodes and workload. As long as the number of these nodes is increased proportionally to the workload increase, the response time remains equal.

The scalability properties of this architecture appear to depend on more factors than the workload parameters, such as the number of online users. When these factors remain constant, as assumed, this architecture scales sublinearly. This means that this architecture is able to handle additional load without needing a linearly proportional number of extra machines.

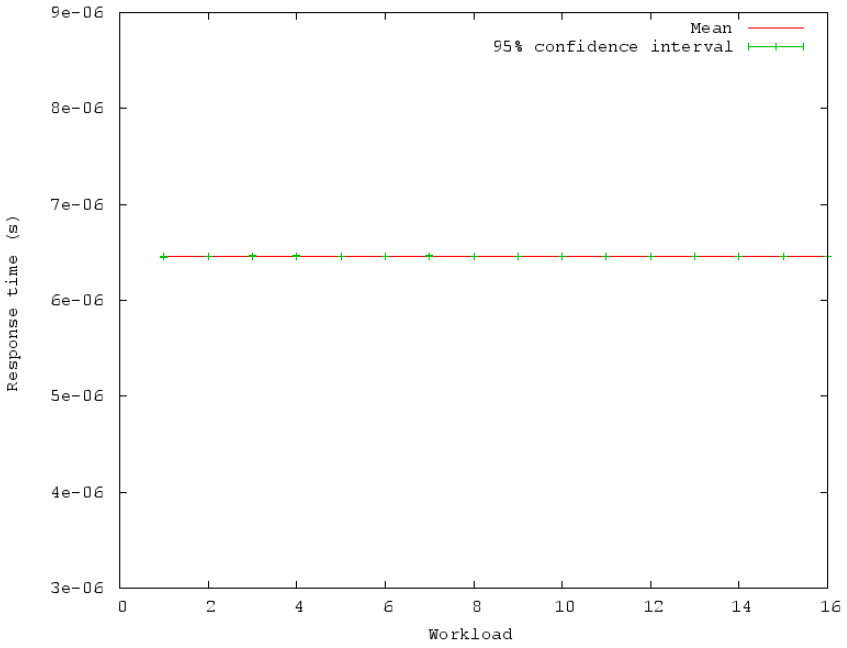


Figure 5.17: Response time of message sending in architecture 2

5.4 Architecture 3: Presence subscriptions

Presence subscriptions are used in the Windows Live Messenger architecture (see Section 2.6.3). The difference between this approach and the approach used in the current architecture and the first proposal is that application nodes subscribe to presence updates at database nodes. This reduces the amount of network traffic.

5.4.1 Overview

Figure 5.18 gives an overview of the subscription-based architecture. Users can connect to any of the *application nodes*. Each *database node* stores subscriptions and presence information for a portion of the user base. Each database node and application is a single machine. There are no master or slave nodes in this architecture.

When a user logs in, the application node sends a subscription to each database node that stores information for a user in the contact list. Whenever one of those users updates their presence, the database node uses the list of subscriptions to forward the presence update to the application nodes. In this way, presence is only forwarded to

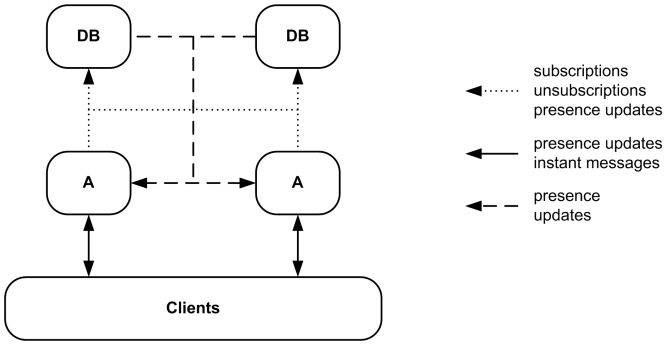


Figure 5.18: Overview of subscription-based architecture

nodes that actually need this information. Application nodes cache presence information for their users' contacts. The subscription mechanism ensures that this cache always contains up to date information. Instant messages are forwarded directly between application nodes, similar to the current architecture. While in that architecture and in the first proposal each instant message meant a lookup in the presence database, in this architecture application nodes can use their internal cache. Presence subscriptions are removed when a user logs out.

This architecture has one type of node less in comparison with the previous two proposals. However, the introduction of a cache on the application node and a subscription storage on the database node, both are relatively more complex than comparable nodes of other architectures.

The subscription approach aligns well with XMPP, as it also used in federation between XMPP domains for propagation of presence updates. Hyves does not yet support federation, but using subscriptions internally gives the opportunity to add this functionality at a lower cost. In comparison to the technique of replication in the current architecture, one could argue that the subscription mechanism is a specialized form of replication where only the updates are replicated that are of interest to users.

5.4.2 Scalability and remaining bottlenecks

This architecture can be scaled by increasing the number of application nodes and the number of the database nodes. The number of database nodes increases with the number of presence updates, logins and logouts. The login and logout actions put a kind of load on the database nodes not seen in other architectures, as these

actions cause subscriptions to be created and destroyed. Adding new database nodes requires special procedures, even when using consistent hashing. Presence information and subscriptions have to be transferred from the old database node to the new node. Application nodes need to be informed of the new number of database nodes and need to adjust the parameters of the consistent hashing algorithm.

The number of application nodes depends on the load caused by all actions combined. Adding new application nodes is as easy as with the current architecture, because database nodes are automatically informed of new application nodes when new subscriptions arrive.

In comparison with the first proposal, the bottleneck of an exploding number of slaves is eliminated. However, the bottleneck of network consumption remains. In comparison with the second proposal, this architecture does not limit the amount of presence information stored per user. Also, the load on the system does not depend on the total number of users.

5.4.3 HIT model

Figure 5.19 shows the top layer of the model. It shows how the actions are mapped to services of the application nodes and database nodes. The *propagpresence* service is responsible for propagating the presence updates from the database nodes to the clients, via the application nodes. Similarly to the database replication process of the current architecture, this process runs asynchronously from the main action process.

Figure 5.20 gives an overview of the application nodes. It shows how the *subscribe* and *unsubscribe* calls to the database nodes are embedded in the *login* and *logout* services. It also shows how presences are stored in the local storage facility *presencestorage*. The *sendmessage* service uses this storage to find the session information of the destination user. The *propagpresence* service represents the processing of an incoming presence update from a database node. The model of the database nodes is given in Figure 5.21. It shows the services offered to the application nodes. Each database node contains storages for presence information and subscriptions.

Table 5.3 contains the additional parameters used by this model. The average size of a presence update packet is assumed to be around the size of a presence update query in the current architecture. Subscription and unsubscription messages contain at least identifiers for the subscribing and subscribed user and the address of the application

node to which updates must be sent. Including protocol headers, this information can be stored in one hundred bytes.

Workload parameters		
numap	Number of application nodes	
numdb	Number of database nodes	
Architecture parameters		
avgpresenceupdatesize	Average size of a presence update packet (bytes)	500
avgsubscribesize	Average size of a subscription request (bytes)	100
avgunsubscribesize	Average size of a unsubscription request (bytes)	100

Table 5.3: Additional model parameters for architecture proposal 3

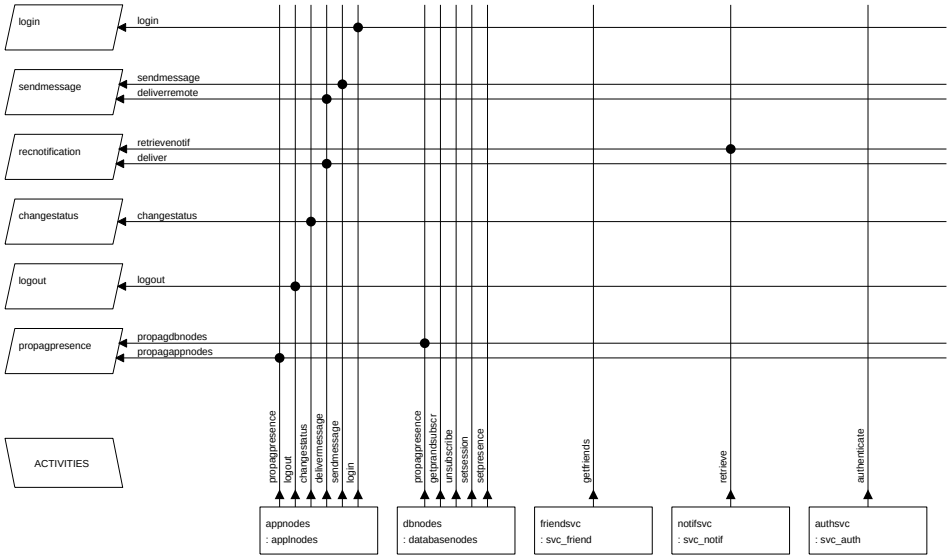


Figure 5.19: Top layer of model of architecture proposal 3

5.4.4 Analysis

Similar to the other two proposals, we use our HIT model to analyse the scalability properties of this architecture. Again, we increase the workload linearly while we increase the number of nodes of one type proportionally.

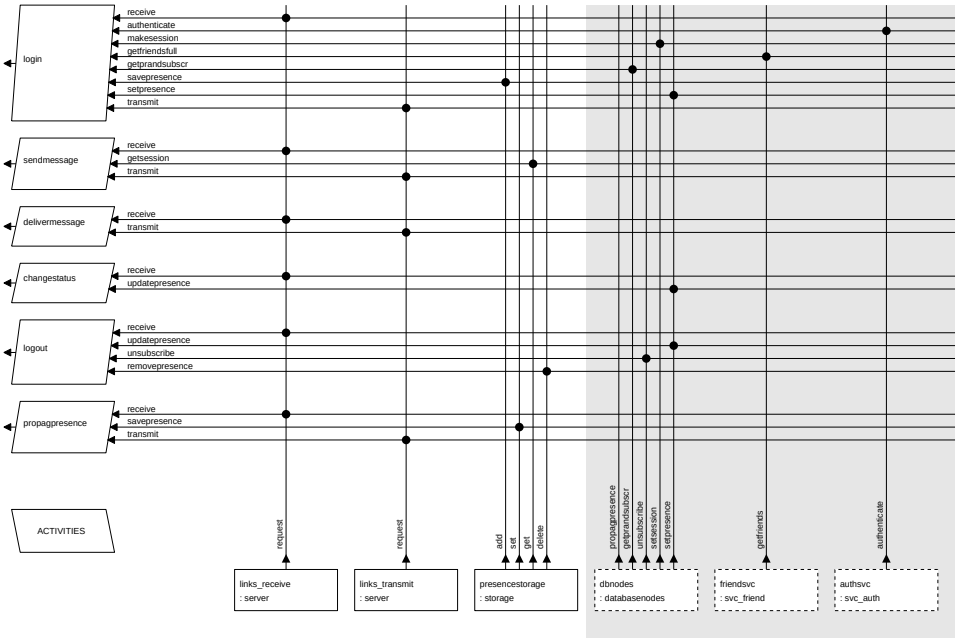


Figure 5.20: Model of application nodes in architecture proposal 3

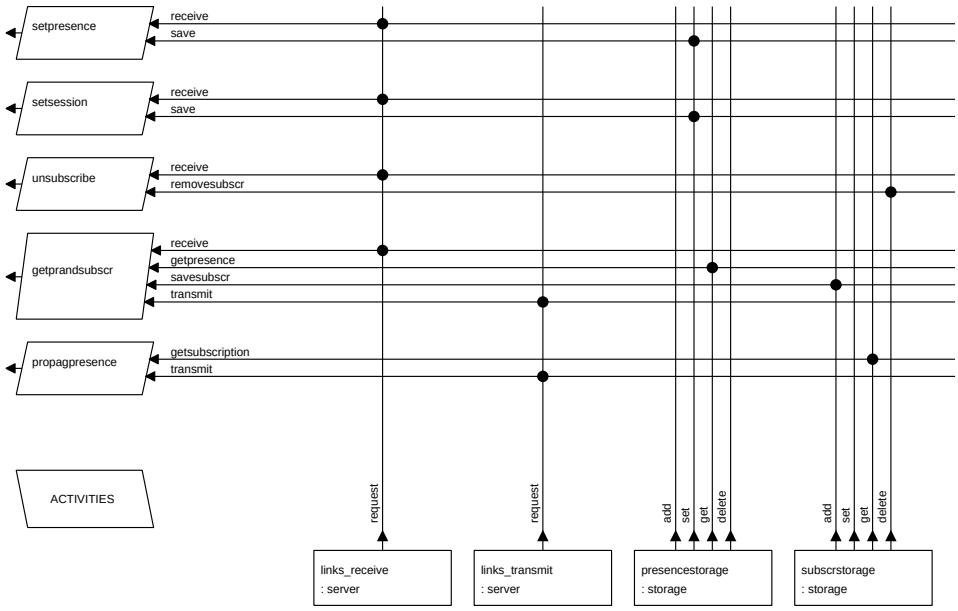
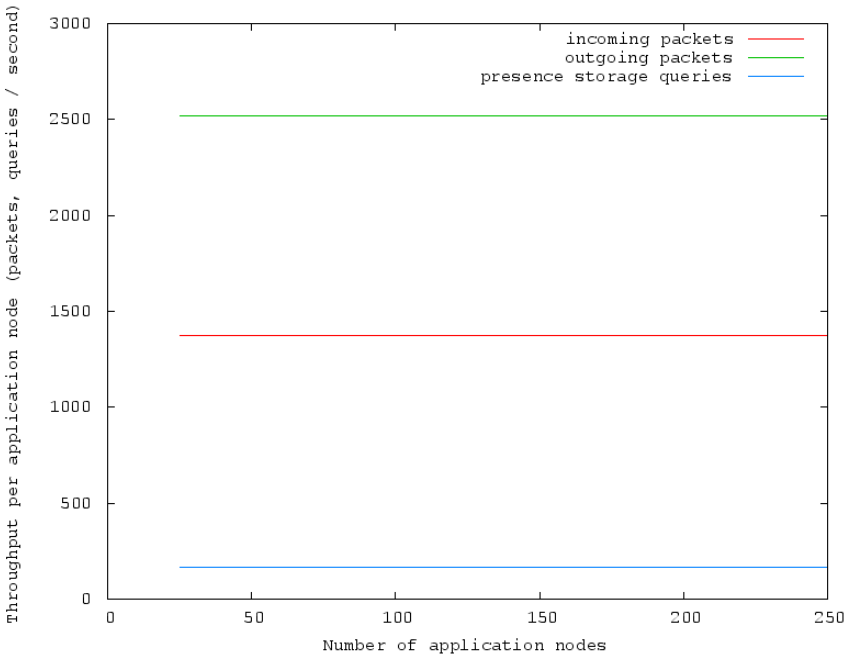
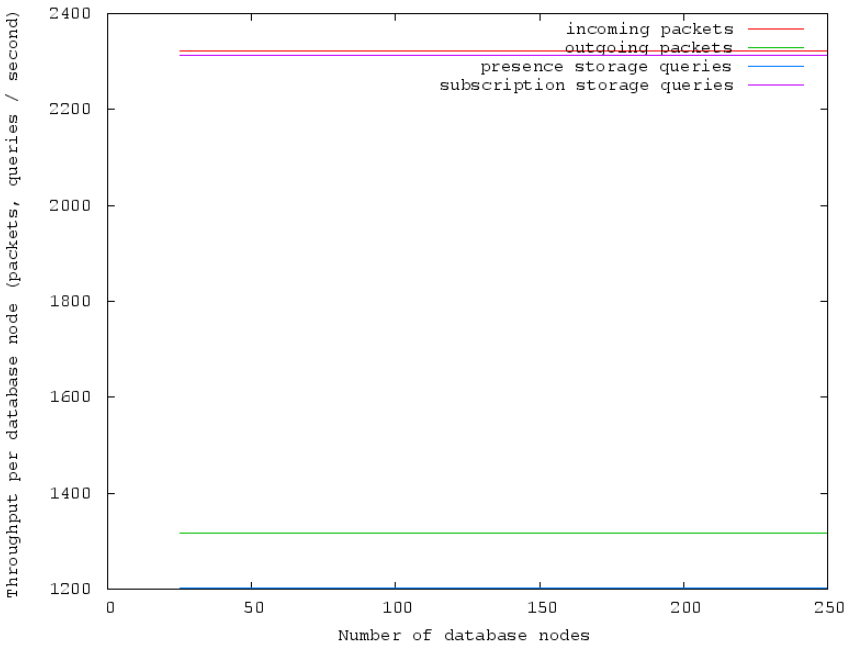


Figure 5.21: Model of database nodes in architecture proposal 3

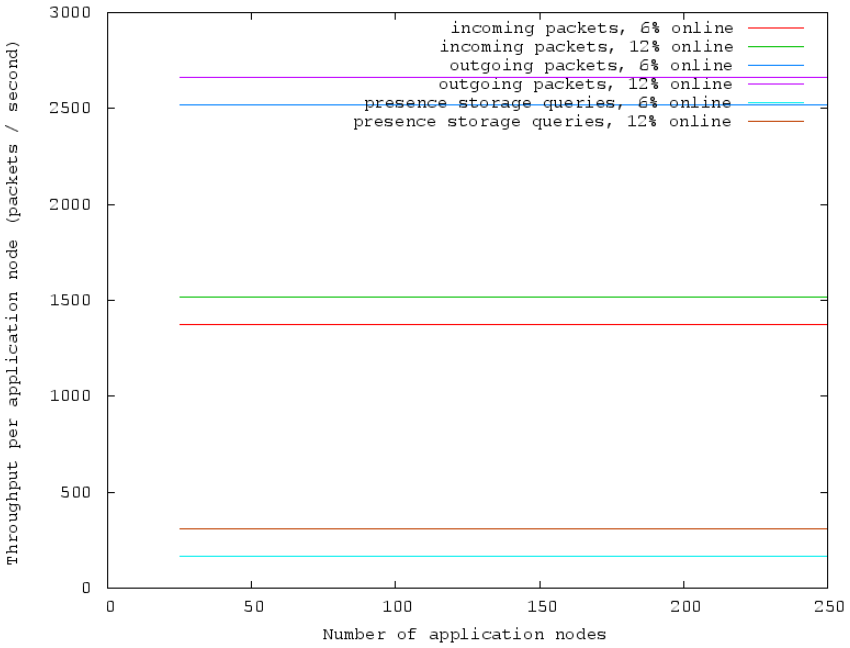


(a) Throughput of application nodes

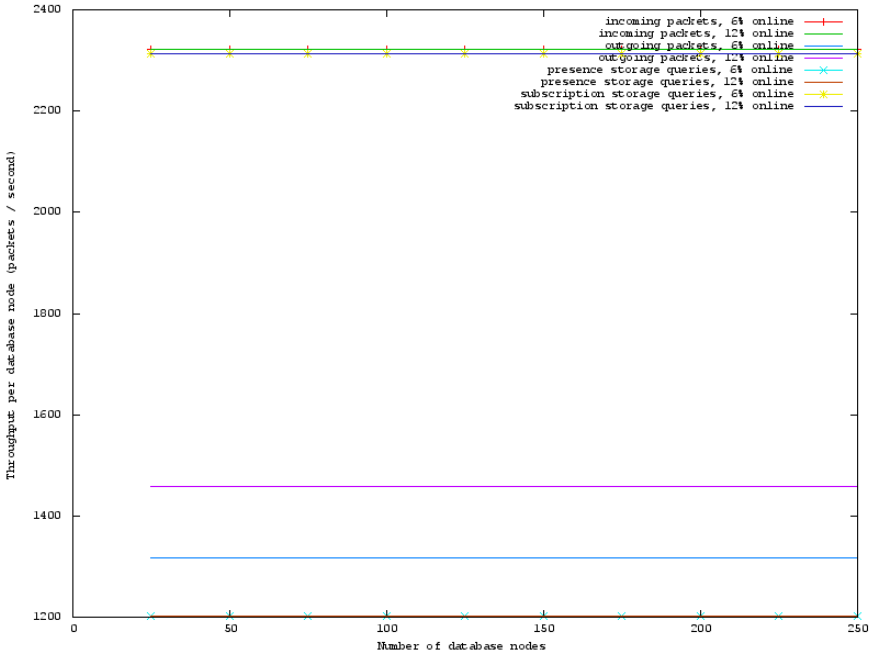


(b) Throughput of database nodes

Figure 5.22: Analysis of proportionally increasing loads in architecture proposal 3



(a) Throughput of application nodes for different parameters



(b) Throughput of database nodes for different parameters

Figure 5.23: Analysis of proportionally increasing loads in architecture proposal 3

Figure 5.22(a) gives the results for the application nodes. Starting with 25 application nodes and one time the current workload, we add 25 application nodes every time the multiple of the workload increases by one. The number of presence nodes is kept constant at 250. These numbers differ from the numbers used when analysing the other proposals, as this model contains more storages, requiring more nodes of both types for the same workload. The throughputs for the incoming and outgoing links are given in the y-axis. The curves of the links and storages are all constant. This indicates that this part of the architecture scales for increasing workloads.

Figure 5.22(b) shows the results of the corresponding experiment for the database nodes. Here, the number of application nodes is kept constant at 250 while the number of database nodes increases by 25. Each database node contains a separate storage for the presence information and the subscriptions. The graph shows that the subscription storage is accessed about two times as frequently as the presence storage. Each curve is constant, indicating good scalability of the database nodes.

A repetition of these experiments for doubled values of *onlinepct* shows that this architecture is only to small extent affected by changes in this parameter. Figure 5.23(a) shows that traffic and storage queries in the application node increase by only a small amount. Figure 5.23(b) shows that only the throughput of the outgoing link increases slightly.

Figure 5.24 shows the response time of sending an instant message. The x-axis shows the number of application and database nodes, which is increased proportionally to the workload. The y-axis shows the latency in seconds. It shows that the latency increases for a while, but approaches a constant value, similar to the first architecture. The message sending process for these two architectures is the same, which explains the similarity. However, this architecture does not need a database lookup for instant message exchange such that the response times are lower than those of the first architecture.

The results of these experiments prove that the components of this architecture scale linearly.

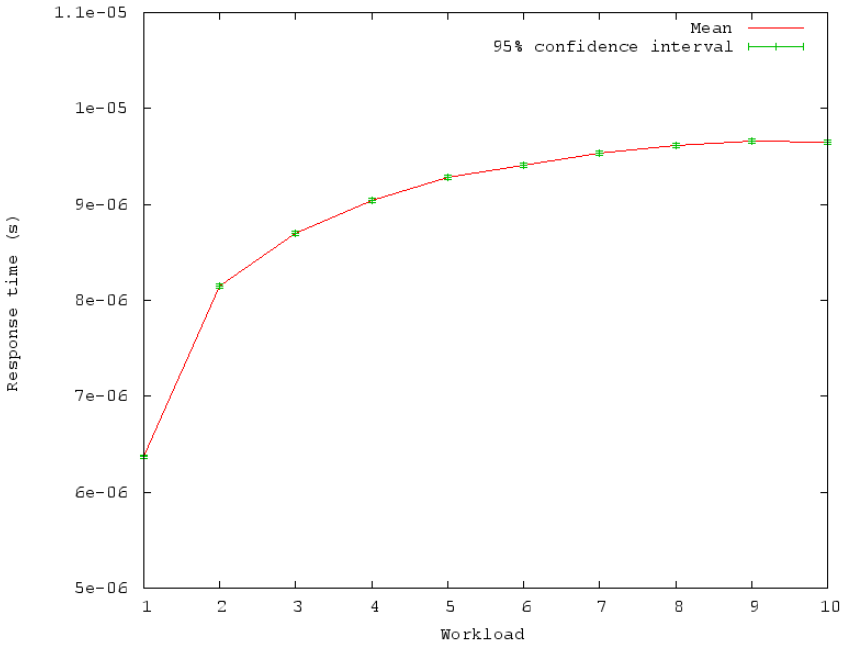


Figure 5.24: Response time of message sending in architecture 3

5.5 Comparative analysis

In this section, we compare the architectures on user-oriented and operator-oriented measures. First, we compare architectures on latency experienced by users. After that, we compare them by the number of machines necessary to handle a workload, a measure useful to the system operators.

5.5.1 Latency of message and presence propagation

The important latency measures are the time between sending a message by one user and receiving that message by another user and updating presence information and reception of that update at each online contact. As explained in Section 4.4.3, the models we developed focus solely on scalability with respect to resource consumption. They lack the necessary detail to estimate latency accurately. However, by comparing the processes in each architecture to handle messages and presence updates we find out which architecture would give the lowest latencies in comparison to the others.

Figure 5.25 gives a view of these processes for each architecture. It shows the flow

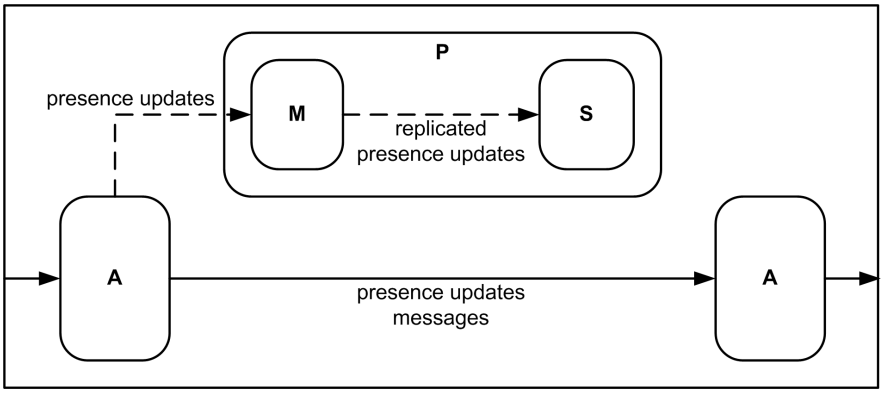
of messages between nodes of an architecture in response to messages and presence updates. For each figure, the incoming arrow at the left represents the communication link between the sender of a message or presence update and the application node to which the sender is connected. The outgoing arrow at the right side represents the link to the recipient of a message or a presence update. Assuming that users are uniformly connected to the available application nodes, the chance that the sender and reciever are connected to the same node is $\frac{1}{numap}$. In the following, we assume that this is not the case, as this provides the worst-case scenario for latency and it is the most probable scenario for large *numap*.

In the first architecture proposal messages and presence updates flow directly from the senders' application node to the receivers' application node. The dashed line indicates the process of storing presence information persistently, which does not influence the propagation of presence updates to online contacts. In the second archicture, messages pass the senders' application node and the receivers' channel node, giving the same number of intermediate nodes as the first architecture. However, presence information is delayed by at most thirty seconds between the channel node and the presence node and by at most 120 seconds between the presence node and the receivers' application node, giving an average latency of 75 seconds. The last architecture processes messages identically to the first architecture, yielding a similar latency. Presence information is propagated by database nodes, which have to look up subscriptions in their storages. This increases the latency in comparison to the first architecture.

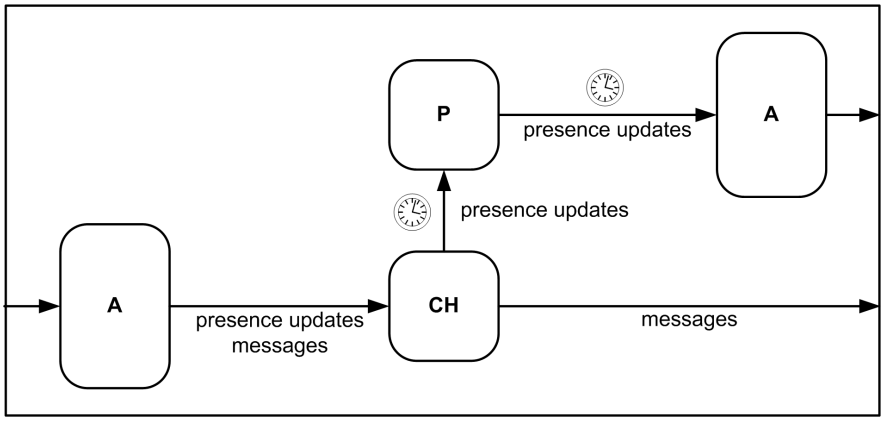
In all architectures, messages only pass two intermediate nodes and experience no forced delays. For presence propagation, the first architecture provides the lowest latency, making this architecture the best for users.

5.5.2 Number of machines

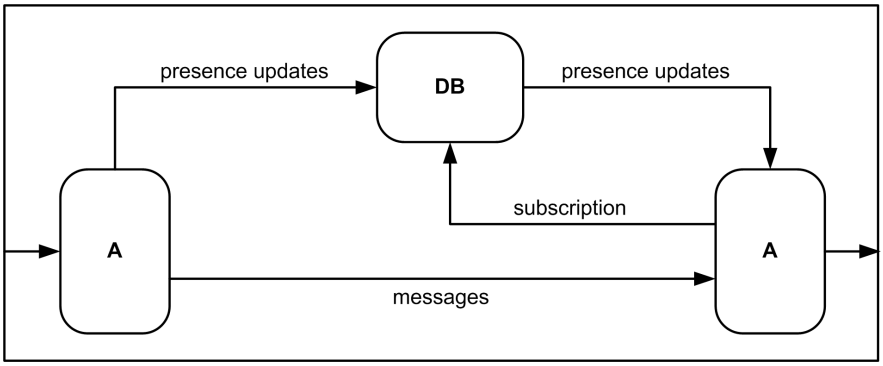
We use the analysis method detailed in Section 4.4.3 to compare the architecture proposals on the number of machines needed to support increasing workloads. Recall that this method poses a very strict requirement on each architecture: for each workload, the utilisation at each resource must be at most the utilisation of that resource for the original workload. The experiment is run with workloads ranging from two times the current workload up to ten times the current workload.



(a) Architecture 1: partitioning



(b) Architecture 2: batched presence updates



(c) Architecture 3: presence subscriptions

Figure 5.25: Flow of presence updates and messages

The results of this experiment are given in Table 5.4, 5.5 and 5.6. The total number of machines is plotted in Figure 5.26. The tables show for each architecture the number of machines necessary for each separate type of node. The column names refer to the input parameters of the respective models. For architecture 1, the relation between the number of nodes is complex. For instance, when adding more slaves to cope with additional read load, the additional replication traffic increases the load on the master. To decrease this to the required level, partitions have to be added, increasing the load on the application nodes. This complexity shows in the analysis results: there is clearly a superlinear increase in number of machines, but a relation between workload and machine count is hard to deduce.

The analysis shows that the second architecture and the third architecture scale linearly. The difference between the two is that for the third architecture the required number of machines is a constant multiple of the workload factor, while for the second architecture there is a base number of machines required, independent of the workload. This is deduced from number of presence servers ($numpr$), which remains fairly constant. This is a direct result from the aggregation of presence updates in this architecture. Recall that the load generated by the distribution of presence updates is unrelated to the number of presence updates per second. This property is unique for this architecture.

From these results we can conclude the following: the first architecture is not scalable. The second architecture is scalable, with the reservation that the number of machines is only partially determined by the workload. The third architecture is scalable in the most strict sense of the word: the number of machines is related linearly to the workload.

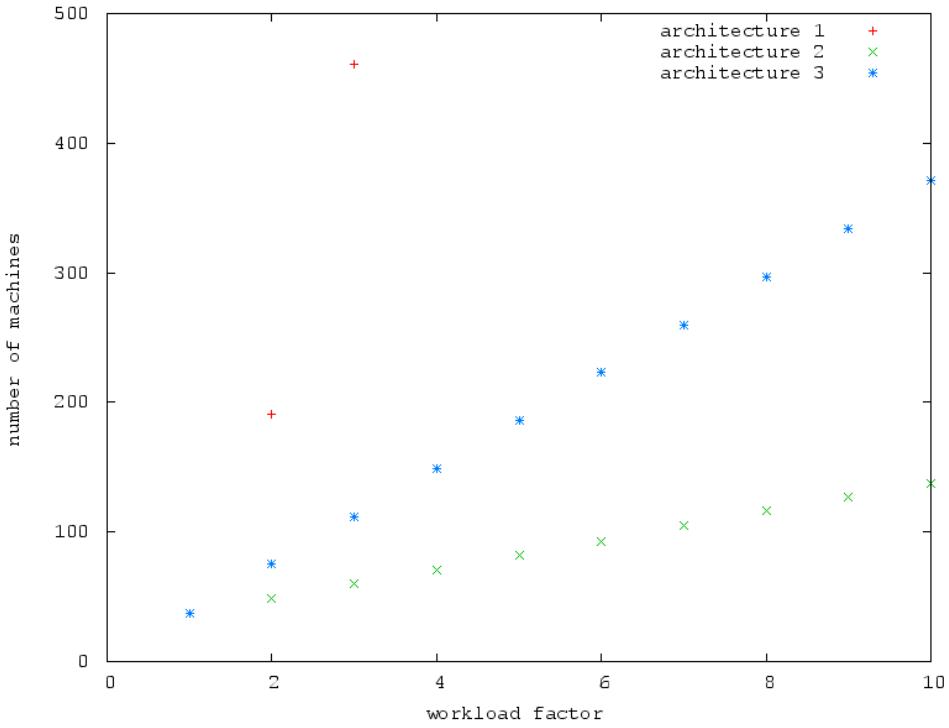


Figure 5.26: Machine analysis

workload	numap	numspp	numpar	total
1	7	4	6	37
2	23	7	21	191
3	51	9	41	461
4	102	12	72	1038
5	179	14	110	1829
6	286	17	153	3040
7	460	19	219	4840
8	625	22	264	6697
9	978	24	375	10353
10	1169	27	405	12509

Table 5.4: Machine analysis on architecture 1

workload	numap	numch	numpr	total
1	7	5	25	37
2	12	10	27	49
3	17	15	28	60
4	22	20	29	71
5	27	25	30	82
6	32	30	31	93
7	37	35	33	105
8	42	40	34	116
9	47	45	35	127
10	51	50	36	137

Table 5.5: Machine analysis on architecture 2

workload	numap	numdb	total
1	8	29	37
2	17	58	75
3	25	87	112
4	33	116	149
5	41	145	186
6	49	174	223
7	57	203	260
8	65	232	297
9	73	261	334
10	81	290	371

Table 5.6: Machine analysis on architecture 3

6

Conclusion

We summarise the results we have made to reach the goals of this research, as defined in Chapter 1.

We introduced a suitable modelling and analysis approach for scalability analysis. It abstracts from the unnecessary performance aspects of the architecture and focuses solely on the relation between workload and the use of databases and network links. Using this approach, we found that the current architecture does not scale in almost all of its parts: the database master nodes do not scale as there is no possibility of adding machines to share the load. We found that the database component will be the first to reach its limit for increasing loads.

Furthermore, we presented three alternative architectures. The first is the result of applying partitioning to the current architecture without changing other aspects. This architecture appears to be scalable in a sense that it is possible to keep adding equipment, but the relation between the amount of equipment needed and the capacity is not linear. The second approach uses batched presence updates to make it more resilient to workload increases. A downside is that it is more heavily influenced by other factors, such as the total number of users and the fraction of active users, than other architectures. In addition to that, it also limits the amount of presence information that can be stored, whereas the other architectures do not have such a fundamental limit. The third architecture uses subscriptions for presence propagation. With only two types of nodes, one less compared to the other alternatives, it is also simpler to implement and maintain.

Comparative analysis shows that the third alternative has a strictly linear relation between workload and utilisation. This relation is also linear for the second architecture, but here the utilisation is only partially determined by the the workload. The first alternative has the worst scalability in this comparison, its machine to workload ratio increases dramatically for larger workloads. At the same time, this architecture shows the lowest complexity in forwarding messages and presence updates and response time analysis shows that the response time of message forwarding approaches a constant value when the architecture is scaled. The third architecture has the same latency for messages, but a higher latency for presence updates, due to the subscription mechanism. The second architecture has a built-in delay in presence propagation, making this architecture the slowest presence propagator. The response time of message propagation is constant, regardless of the scale of the architecture.

These findings lead us to recommend the subscription-based architecture as the best architecture for Instant Messaging & Presence services. The evolutionary improved architecture might be used as a short term solution if scalability problems arise, but the model shows that it does not scale in the long run. The aggregated presence updates architecture is also scalable, but is less favourable, given the degradation in quality of service for presence propagation.

Future work on this topic can be done in a multitude of directions. On the practical aspect of this thesis, comparing the performance of the architectures by simulation or prototyping can provide additional insight in the benefits of each architecture. Also, research can be aimed at just one architecture and discover the precise resource requirements of this architecture to aid resource planning. On the theoretical side of this thesis, better tool support for the analysis of computer architectures needs to be developed. HIT and HITGRAPHIC were sufficient, but specific enhancements for models of scalable architectures would decrease the analysis effort. Specifically, modelling multiple identical components is only possible using simulation, while it could probably be analysed analytically if HIT would support this. The HITGRAPHIC representation of a HIT model makes modelling easier, but this can be further enhanced by focusing more on relations between components. Cycles in the component hierarchy are not allowed, which prevented a straight-forward implementation of the current architecture.

6.1 Related work

Queueing networks have been used to model computer communication networks ever since the first computers were connected. (Wong, 1978) describes the relation between delay and throughput in a simple computer network using a queueing network model.

Deriving a queueing network for a CORBA system to predict performance has been described in (Harkema et al., 2004). The model and two sets of input parameters were compared to measurements on a functional system. According to the authors, the performance predictions based on the model accurately match the results from the lab experiments.

A web server architecture was modelled using queueing networks in (Slothouber, 1995). The average file size, the number of web servers, the speed of the web servers and the speed of the network were related to the response time. Analysis of the model revealed that the response time increases asymptotically after the load increased beyond some point. This point is closely related to the average size of the served files. The paper also discusses the gains of several architectural improvements, such as adding a web server or increasing the network bandwidth, depending on the kind of load on the architecture.

A general approach to analyse distributed architectures using Markov chains and queueing networks is given in (Sharma et al., 2005). It involves modelling the system using a Discrete Time Markov Chain (DTMC). The resulting total service requirements are used to create a closed Product Form Queueing Network (PFQN). The PFQN is processed by the SHARPE software package, to find bottlenecks and estimate the merits of architectural modifications.

A comparison between two architectures using queueing networks is performed in (Liu and Gorton, 2005). A client - server architecture is studied, where the server is composed of a front-end request handler and a back-end message based request processor. The input variables for the queueing stations are determined using benchmarking on a physical implementation. Latency and throughput are calculated for three architectural variants and compared to measured values using a prototype implementation. The error margin of the performance prediction proved to be within 15%.

A similar problem is discussed in (Lladó and Harrison, 2000), which investigates the concurrency limits of an Enterprise Java Bean container. The components of the

architecture are modelled separately using Markov Chains. The models are aggregated in a Flow Equivalent Server (FES) model. The FES model is compared to a simulation model that was created with the tool QNAP2. The model very well approximates both configurations.

The SIP Instant Messaging and Presence Leveraging Extensions working group of the Internet Engineering Task Force develops an IM&P extension to the Session Initiation Protocol (SIP). The working group has published an analysis of the message load caused by increasing workloads in various inter-domain scenarios (Houri et al., 2008). Inter-domain presence exchange uses a publish/subscription mechanism similar to the third alternative architecture discussed in this thesis. Although the paper does not discuss architectural properties, the presented numbers indicate that the presence extensions for SIP will cause a high load in terms of number of messages, network bandwidth, state management and CPU. The authors conclude by recommending research on a optimised server-to-server protocol for inter-domain scalability.

References

Abello, J., Pardalos, P. and Resende, M.G.C. *On maximum clique problems in very large graphs. External memory algorithms*, pages 119–130, 1999.
URL http://portal.acm.org/citation.cfm?id=327766.327783 39

Barroso, L.A., Dean, J. and Hölzle, U. *Web Search for a Planet: The Google Cluster Architecture. IEEE Micro*, 23(02):22–28, 2003. ISSN 0272-1732.
URL http://dx.doi.org/10.1109/MM.2003.1196112 9

Beilner, H., Mäter, J. and Weißenberg, N. *Towards a Performance Modelling Environment: News on HIT.* In *Modelling Techniques and Tools for Computer Performance Evaluation*, pages 57–75. Plenum, 1989. 30

Beilner, H., Mater, J. and Wysocki, C. *The Hierarchical Evaluation Tool HIT.* In D. Potier and R. Puigjaner, editors, *Short Papers and Tool Descript. of 7th Int. Conf. on Modelling Techniques and Tools for Computer Perf. Evaluation*, pages 6–9. 1994.
URL http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.36.516 30

Bondi, A.B. *Characteristics of scalability and their impact on performance.* In *WOSP '00: Proceedings of the 2nd international workshop on Software and performance*, pages 195–203. ACM Press, New York, NY, USA, 2000. ISBN 158113195X.
URL http://dx.doi.org/10.1145/350391.350432 6

Büttner, M. et al. *HI-SLANG Reference Manual for the Hierarchical Evaluation Tool HIT.* Universität Dortmund, Informatik IV, 3.6.00 edition, 1999. 31

Erlang, A.K. *The Theory of Probabilities and Telephone Conversations. Nyt Tidsskrift for Matematik*, 20(B):33–39, 1909. 29

Gilbert, S. and Lynch, N. *Brewer's conjecture and the feasibility of consistent, available, partition-tolerant web services. SIGACT News*, 33(2):51–59, 2002. ISSN 0163-5700.
URL http://dx.doi.org/10.1145/564585.564601 9

Haerder, T. and Reuter, A. *Principles of transaction-oriented database recovery. ACM Comput. Surv.*, 15(4):287–317, 1983. ISSN 0360-0300.
URL http://dx.doi.org/10.1145/289.291 8

Haines, S. *Pro Java EE 5 Performance Management and Optimization.* Apress, 1 edition, 2006. ISBN 1590596102.
URL http://www.amazon.com/exec/obidos/redirect?tag=citeulike07-20&path=ASIN/1590596102 7

Harkema, M. et al. *Middleware Performance: A Quantitative Modeling Approach.* In *Symposium on Performance Evaluation of Computer Telecommunication Systems*, pages 733–742. 2004.
URL http://www.scs.org/getDoc.cfm?id=2885 2, 95

Haverkort, B.H. *Performance of Computer Communication Systems*, chapter 4.7, pages 80–83. John Wiley & Sons, 1998. ISBN 0-471-97228-2. 43

Hill, M.D. *What is scalability? SIGARCH Comput. Archit. News*, 18(4):18–21, 1990. ISSN 0163-5964.
URL http://dx.doi.org/10.1145/121973.121975 6

Houri, A. et al. *Presence Interdomain Scaling Analysis for SIP/SIMPLE.* Internet Draft, 2008.
URL http://tools.ietf.org/html/draft-ietf-simple-interdomain-scaling-analysis-05 96

Jabber, I. *Jabber XCP Performance Tests - From 100,000 To One Million Concurrent Users.* 2007.
URL http://www.jabber.com/media/Jabber_XCP_Performance_100000_1Million.pdf 13

Jogalekar, P. and Woodside, M. *Evaluating the scalability of distributed systems. Parallel and Distributed Systems, IEEE Transactions on*, 11(6):589–603, 2000.
URL http://dx.doi.org/10.1109/71.862209 6

REFERENCES

Karger, D. et al. *Consistent hashing and random trees: distributed caching protocols for relieving hot spots on the World Wide Web.* In *STOC '97: Proceedings of the twenty-ninth annual ACM symposium on Theory of computing*, pages 654–663. ACM Press, 1997. ISBN 0897918886.
URL http://dx.doi.org/10.1145/258533.258660 11

Karger, D. et al. *Web caching with consistent hashing.* In *WWW '99: Proceedings of the eighth international conference on World Wide Web*, pages 1203–1213. Elsevier North-Holland, Inc., New York, NY, USA, 1999.
URL http://dx.doi.org/10.1016/S1389-1286(99)00055-9 11

Kegel, D. *The C10K problem.* Technical report, Kegel.com, 2006.
URL http://www.kegel.com/c10k.html 12

Kendall, D.G. *Stochastic Processes Occurring in the Theory of Queues and their Analysis by the Method of the Imbedded Markov Chain. The Annals of Mathematical Statistics*, 24(3):338–354, 1953. ISSN 00034851.
URL http://dx.doi.org/10.2307/2236285 28

Letucky, E. *Facebook Chat.* 2008.
URL http://www.facebook.com/note.php?note_id=14218138919 14

Little, J.D.C. *A Proof for the Queuing Formula: L= λ W. Operations Research*, 9(3):383–387, 1961. ISSN 0030364X.
URL http://dx.doi.org/10.2307/167570 29

Liu, Y. and Gorton, I. *Performance Prediction of J2EE Applications Using Messaging Protocols.* In *Component-Based Software Engineering*, volume 3489/2005, pages 1–16. Springer Berlin / Heidelberg, 2005.
URL http://dx.doi.org/10.1007/11424529_1 95

Lladó, C.M. and Harrison, P.G. *Performance evaluation of an enterprise JavaBean server implementation.* In *WOSP '00: Proceedings of the 2nd international workshop on Software and performance*, pages 180–188. ACM, New York, NY, USA, 2000. ISBN 1-58113-195-X.
URL http://dx.doi.org/10.1145/350391.350429 95

Nicola, M. and Jarke, M. *Performance modeling of distributed and replicated databases. Knowledge and Data Engineering, IEEE Transactions on*, 12(4):645–672, 2000.
URL http://dx.doi.org/10.1109/69.868912 26

Peddemors, A., Lankhorst, M. and de Heer, J. *Presence, Location, and Instant Messaging in a Context-Aware Application Framework.* In *Mobile Data Management*, pages 325–330. Springer Berlin / Heidelberg, 2003.
URL http://dx.doi.org/10.1007/3-540-36389-0_22 8

Piro, C. *Chat Stability and Scalability.* 2009.
URL http://www.facebook.com/note.php?note_id=51412338919 14

Reisinger, D. *How Twitter could be worth nothing in a year.* 2008.
URL http://news.cnet.com/8301-13506_3-9981717-17.html 1

Saint-Andre, P. *Extensible Messaging and Presence Protocol (XMPP): Core.* 2004a.
URL http://tools.ietf.org/pdf/rfc3920.pdf 23

Saint-Andre, P. *Extensible Messaging and Presence Protocol (XMPP): Instant Messaging and Presence.* 2004b.
URL http://tools.ietf.org/pdf/rfc3921.pdf 23

Sharma, V.S., Jalote, P. and Trivedi, K.S. *Evaluating Performance Attributes of Layered Software Architecture.* In *Component-Based Software Engineering*, volume 3489/2005, pages 66–81. Springer Berlin / Heidelberg, 2005.
URL http://dx.doi.org/10.1007/11424529_5 95

Skeen, D. and Stonebraker, M. *A Formal Model of Crash Recovery in a Distributed System. Software Engineering, IEEE Transactions on*, SE-9(3):219–228, 1983.
URL http://ieeexplore.ieee.org/xpls/abs_all.jsp?arnumber=1703048 9

Slothouber, L.P. *A Model of Web Server Performance.* In *5th International World Wide Web Conference.* 1995. 2, 95

Smolander, K. *Four metaphors of architecture in software organizations: finding out the meaning of architecture in practice.* In *Empirical Software Engineering, 2002. Proceedings. 2002 International Symposium n*, pages 211–221. 2002.
URL http://dx.doi.org/10.1109/ISESE.2002.1166942 5

Torre, C. *Windows Live Messenger - What. How. Why.* 2006.
URL http://channel9.msdn.com/posts/Charles/Windows-Live-Messenger-What-How-Why/ 15

Trivedi, K.S. *Probability and Statistics with Reliability, Queueing, and Computer Science Applications, 2nd Edition.* Wiley-Interscience, 1 edition, 2001. ISBN 0471333417.
URL http://www.amazon.com/exec/obidos/redirect?tag=citeulike07-20&path=ASIN/0471333417 27

Vogels, W. *Eventually consistent. Communications of the ACM*, 52(1):40–44, 2009. ISSN 0001-0782.
URL http://dx.doi.org/10.1145/1435417.1435432 9

Wong, J.W. *Queueing Network Modeling of Computer Communication Networks. ACM Comput. Surv.*, 10(3):343–351, 1978. ISSN 0360-0300.
URL http://dx.doi.org/10.1145/356733.356740 2, 95