Postina: A Publish/Subscribe Middleware Designed for Massively Multiplayer Games

Master Thesis 2007-2008

`http://postina.zindel.org`

Supervisors: Jörg Kienzle, Alexandre Denault and Ulrich Ultes-Nitsche

Dominik Zindel
dominik [at] zindel.org

School of Computer Science
McGill University, Montréal, Québec
and
Department of Informatics
University of Fribourg, Switzerland

April 15, 2008

Abstract

Postina is a network middleware designed for massively multiplayer online games (MMOG). It combines publish/subscribe functionalities with direct messaging, a feature of critical importance for MMOGs.

In MMOGs, numerous messages such as state updates are sent to different clients. While some messages must be multicast to a large group of clients, other messages are private and sent to a single client only. Using a traditional client-server approach limits the number of simultaneous players, a property that is undesired in MMOGs. A potential solution would be to use publish/subscribe systems which are designed for scalability. Pure publish/subscribe systems, however, do not provide any possibility to send a message directly to a single client as the peers do not have any knowledge of the network topology.

In this thesis we first study different publish/subscribe systems and the features they offer to choose an appropriate middleware providing the required functionalities for MMOGs. Additionally, we design Postina, an API for network layers in MMOGs that offers a convenient interface combining publish/subscribe and direct messaging. We then implement a version of Postina using Scribe, a topic-based publish/subscribe system built on top of the distributed hashtable Pastry. To fulfil the requirements of MMOGs, we add extra features such as reliable direct messaging and the capability of issuing subscriptions for other clients. This version of Postina is then integrated into Mammoth, the massively multiplayer game research framework developed at McGill University. Finally, the efficiency of the new network layer is tested to determine the maximum number of simultaneous players.

This thesis was written at McGill University (Montréal, Québec) under the supervision of Professor Jörg Kienzle and Alexandre Denault (PhD candidate) as part of the fulfilment of the requirements of the Master of Science in Computer Science at the University of Fribourg (Switzerland). The thesis was co-supervised by Professor Ulrich Ultes-Nitsche at the University of Fribourg (Switzerland).

Abrégé

Postina est un intergiciel ("middleware") pour la communication de réseau dans des jeux en ligne massivement multijoueur (MMOG) qui combine les avantages des systèmes publish/subscribe ("publier et s'abonner") avec la possibilité d'envoyer un message directement à un client spécifié, une fonctionnalité d'importance primordiale pour les MMOGs.

Dans des MMOGs, de nombreux messages de mises à jour sont expédiés à plusieurs participants. Tandis que certains de ces messages sont envoyés à un groupe de clients, d'autres ciblent un seul participant. L'utilisation d'une architecture client/serveur traditionnelle n'est pas appropriée car elle limite le nombre de joueurs qui peut être connecté simultanément, une caractéristique indésirable pour des MMOGs. Cependant, un système publish/subscribe permettrait, par sa conception, d'atteindre un degré maximal d'extensibilité. Malheureusement, les différents clients dans un système publish/subscribe ne connaissent pas tous les autres participants du système. Comme conséquence, l'envoi direct d'un message d'un particulier à un seul client n'est pas possible avec un système publish/subscribe dans sa forme pure.

Dans ce rapport nous étudions d'abord plusieurs systèmes de publish/subscribe existants et leurs fonctionnalités. Cette étude est suivie par le choix d'un intergiciel approprié qui met à disposition les fonctionnalités requises par des MMOGs. Ensuite nous développons Postina, qui combine les fonctionnalités des systèmes publish/subscribe avec l'envoi direct d'un message destiné à un seul client. Postina fournit donc une interface de programmation (API) convenable pour la couche réseau dans des MMOGs. En outre, nous implémentons une version de Postina en utilisant Scribe, un système publish/subscribe qui utilise des sujets ("topic-based") et qui se base sur la table de hachage distribuée Pastry. Afin de satisfaire les besoins des MMOGs nous y ajoutons certaines fonctionnalités comme le transfert fiable de messages et la possibilité d'abonner un autre client à un sujet. Cette implantation concrète de Postina est ensuite intégrée dans Mammoth, le cadre d'applications développé à l'université de McGill pour la recherche dans des jeux massivement multijoueur. Cette application concrète permet de tester le nouveau intergiciel et de déterminer le nombre maximal de clients pouvant se connecter simultanément.

Cette thèse a été écrite à l'université McGill (Montréal, Québec) sous la supervision du Professeur Jörg Kienzle et du doctorant Alexandre Denault en vue de l'obtention du Master of Science en Informatique à l'Université de Fribourg (Suisse). La thèse a été co-supervisée par le Professeur Ulrich Ultes-Nitsche de l'Université de Fribourg (Suisse).

Zusammenfassung

Postina ist eine Middleware für die Netzwerkkommunikation in Massen-Mehrspieler-Online-Gemeinschaftsspielen (MMOG). Es verbindet die Möglichkeiten von Publish/Subscribe-Systemen ("veröffentlichen und abonnieren") mit der für MMOGs äusserst wichtigen direkten Zustellung von Meldungen an einen bestimmten Client ("Kunden") im Netzwerk.

In MMOGs muss eine grosse Menge von Meldungen an eine Vielzahl von Clients geschickt werden um deren Status zu aktualisieren. Während ein Teil der Nachrichten mittels Mehrpunktverbindungen an eine grosse Gruppe von Clients geschickt werden muss (Multicast), sind andere Meldungen privat und müssen nur einem einzigen, bestimmten Client zugestellt werden. Während ein traditionelles Client-Server-Modell einfach umzusetzen ist, schränkt es die mögliche Anzahl der gleichzeitig aktiven Spieler stark ein, da alle Benachrichtigungen des MMOG über den zentralen Server laufen. Diese Eigenschaft limitiert den Nutzen des zentralisierten Client-Server-Modells für Benachrichtigungsdienste in MMOGs. Demgegenüber ist ein Publish/Subscribe-System zwar gut skalierbar; da die einzelnen Teilnehmer im Rechnernetz keine Informationen über die anderen Clients besitzen, haben reine Publish/Subscribe-Systeme jedoch den Nachteil, dass der Versand privater Meldungen direkt an einen bestimmten Teilnehmer nicht möglich ist.

In dieser Arbeit untersuchen wir zuerst verschiedene Publish/Subscribe-Systeme und ihre Funktionalitäten um anschliessend eine passende Middleware auszuwählen. Diese muss die Anforderungen von MMOGs erfüllen. Im Anschluss an diese Untersuchung entwickeln wir Postina, eine Programmierschnittstelle (API) für Netzwerkschichten in MMOGs. Postina stellt eine komfortable Schnittstelle zur Verfügung, die die Nutzung sowohl von Publish/Subscribe-Funktionalitäten als auch von direktem Mitteilungsversand ermöglicht. Zusätzlich implementieren wir eine Version von Postina mit Hilfe von Scribe, einem themenbasierten ("topic-based") Publish/Subscribe-System, das auf der verteilten Hashtabelle Pastry basiert. Um die Anforderungen von MMOGs erfüllen zu können, fügen wir weitere Fähigkeiten hinzu. Dazu gehören der zuverlässige direkte Mitteilungsversand sowie die Möglichkeit, andere Clients als Abonnenten eines Themas hinzuzufügen. Diese Version von Postina binden wir anschliessend in Mammoth, dem an der McGill-Universität für die Forschung an Massen-Mehrspieler-Gemeinschaftsspielen entwickelten Framework, ein, um danach die Leistungsfähigkeit der neuen Netzwerkschicht zu testen und so die maximale Anzahl gleichzeitig verbundener Spieler in Mammoth herauszufinden.

Diese Masterarbeit wurde an der McGill-Universität in Montréal (Québec) unter der Betreuung von Professor Jörg Kienzle und dem Doktoranden Alexandre Denault im Rahmen des Studienprogrammes zur Erlangung des Master of Science in Informatik an der Universität Fribourg (Schweiz) erstellt. Die Arbeit wurde von Professor Ulrich Ultes-Nitsche an der Universität Fribourg (Schweiz) mitbetreut.

Acknowledgement

First, I would like to thank Prof. Jörg Kienzle, my supervisor at McGill, and Prof. Ulrich Ultes-Nitsche at University of Fribourg for enabling me to do my Master thesis as a visiting research student. Their support was invaluable during the realisation of my project. Additionally, I would like to thank Alexandre Denault at McGill University for his support, feedback, time, and ideas.

I would also like to thank Alex Cheung at University of Toronto for the help during the evaluation of Padres and Jeff Hoye at Max Planck Institute for Software Systems for providing assistance during the implementation of Postina with Pastry/Scribe. I am also grateful to Samuel Gélineau at McGill University for the fruitful discussions of theoretical issues. I would also like to thank Renzo Caduff and Christian Markus Eichenberger, both at University of Fribourg, for their help during the process of finding an euphonic product name for this project. Additionally, I am very grateful to my aunt Judith Gantenbein-Schenk who has proofread this thesis.

Last but not least I would like to thank my parents for their support during my studies in Fribourg and Montréal.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

1.1 Motivation

Massively Multiplayer Online Games (MMOG) are becoming increasingly popular. In these games, many players are playing simultaneously. As competing players need to have the same knowledge about the current game state, a lot of data has to be transmitted over the network, usually the Internet. Handling the massive amount of data caused by thousands of players is very challenging and achieving scalability still is a challenge for developers of MMOGs.

Mammoth is a massively multiplayer game research framework. The current networklayer uses the client-server paradigm and opens a TCP connection to each client. If a message is sent to n clients, the server sends n messages. This limits the number of active clients. However, the goal of the Mammoth team is to support several thousand simultaneously connected players, which is not possible with a simple client-server paradigm.

With the current client-server architecture in Mammoth it is very challenging to distribute the load on several servers due to the lack of flexibility. For a distributed infrastructure, the network layer should also be based on a peer-to-peer system.

The currently existing infrastructure for communication in Mammoth has worked well until now. However, it cannot cope with the new requirements of Mammoth. Therefore, an alternative, more powerful network layer is needed.

1.2 Task of Thesis

The task of this master thesis is to evaluate different solutions for the network layer of MMOGs to choose the most powerful one and to define a convenient API for MMOGs. This new API will then be implemented and integrated into Mammoth as a proof of concept.

A special focus is directed to publish/subscribe systems as scalability is their strength. This is especially the case when a messages has to be multicast,

that is sent to an entire group of clients interested in this message (but not to all clients in the network) which is often necessary in multiplayer games. In addition to multicasting, direct messaging is necessary for MMOGs. Several publish/subscribe systems are evaluated and finally one is chosen and applied.

The aim of this report is to document the completed thesis and the decisions taken as well as to provide a sufficient amount of information to developers to use Postina. After this introductory chapter, an overview of possible alternative solutions with a special focus on publish/subscribe systems follows. Third, the demands on the network layer for a MMOG are defined. Particular attention is given to the Mammoth project. Fourth, Postina, the new API providing both direct messaging and publish/subscribe, is introduced and implemented using an existing publish/subscribe system. Fifth, a proof of concept is given by integrating Postina into Mammoth, also explaining how to use it within Mammoth. Despite all efforts, the implemented version of Postina is not a perfect framework and thus its limitations are discussed in Chapter 6. To conclude, a summary is given, which is followed by an outlook mentioning possible improvements and a conclusion.

1.3 Used Software, Environment

The entire project has been developed in `Java 1.5` using the IDE `Eclipse 3.3` on a computer with `Gentoo Linux` and `Ubuntu Linux`. The UML diagrams were drawn using `Dia 0.96.1`, profiling was done using `JProfiler 5.1`.

1.4 Schedule

Research and implementation for this master thesis were done from September 2007 to February 2008 at the Software Engineering Lab at the School of Computer Science at McGill University, Montreal. The first month was dedicated to the evaluation of alternatives. After, Pastry/Scribe were tested in some more details before the actual design and implementation work of Postina was done. Next, Postina was integrated into Mammoth, followed by intense testing and evaluation of the new solution.

Chapter 2

Evaluation of Potential Middleware Systems

The aim of this chapter is to first give an introduction to publish/subscribe systems in general but with a special focus on the requirements of MMOGs (see Chapter 3 on page 23). Second, Sub-2-Sub, an autonomous self-organizing peer-to-peer content-based publish/subscribe system, is introduced. Third, the content-based publish/subscribe middleware PADRES, developed at the University of Toronto, is explained. Fourth, Pastry, an overlay and routing network for the implementation of a distributed hash table, is presented. Fifth, SCRIBE, a topic-based publish/subscribe system based on Pastry is introduced. Last, a short comparison of the four systems is given and two of them are selected for a closer evaluation with regard to the requirements of Mammoth.

2.1 Publish/Subscribe

This section shall give a short and general introduction to publish/subscribe systems.

2.1.1 Application Area

A publish/subscribe system can be applied when a large number of subscribers with diverse interests have to be notified about an event or a publication. The decoupling of clients allows for a maximum degree of flexibility and scalability. In general, publish/subscribe systems can be used when asynchronous communication is necessary.

The scalability in publish/subscribe systems relies on the fact that clients do not need global knowledge of the network. Publishers do not know the consumers, and vice versa. Thus, as a disadvantage, direct messaging between different clients is usually impossible in publish/subscribe systems.

2.1.2 Actors

With publish/subscribe systems, three types of actors are important:

Publishers or *producers* submit data as *publication*s or *notification*s.

Subscribers or *consumers* subscribe to publications, submit subscriptions.

Brokers or *event services* are neutral mediators between publishers and subscribers. Brokers are not used in all publish/subscribe systems. However, the lack of brokers breaks the separation between publishers and subscribers and thus limits the scalability. Therefore, all larger publish/subscribe systems use a form of broker.

Publishers and subscribers are both clients. A client can be both a publisher and a subscriber at the same time.

2.1.3 Overview

In a publish/subscribe system, the clients asynchronously exchange notifications, the publisher does not wait for an answer from the subscriber. As they even do not know of each others' existence, there is no direct point-to-point communication between two actors but all interaction is mediated by a set of brokers or propagators. When receiving a publication, these brokers determine the set of matching subscriptions and forward the publication to the subscribers.

2.1.4 Selection Mechanisms

In publish/subscribe systems there exist five different main notification selection mechanisms (see [16] and [13]). The selection mechanism is of prime importance for a publish/subscribe system. In case of insufficient filters unnecessarily broad subscriptions may saturate the network and require additional filtering on the client-side. The following notification selection mechanisms shall be explained briefly: channel-based, subject-based, topic-based, type-based and content-based selection.

Channel-based Selection

Channel-based selection is the oldest and simplest way to select the messages of interest. A set of channels to which consumers can subscribe is defined. Publications are then published with respect to a certain channel and will be forwarded to all subscribers of the chosen channel.

With channel-based selection, the expressiveness is limited. Publications can only be classified with respect to the defined channels. Additionally, channels are very inflexible. If the set of channels is modified, all clients (subscribers and publishers) may have to be modified.

Subject-based Selection

With subject-based selection, publications are released with respect to a certain subject where a subject usually is a dot separated string. This dot separated string (e.g. `market.quotes.NASDAQ`) represents a subject tree.

Due to the tree structure, subject-based selection has a limited expressiveness as additional criteria cannot be defined easily. In addition, subject division is possible in only one dimension. The use of several dimensions makes the subject tree explode due to the repetition of parts of it. Like channel-based selection, subject-based selection is inflexible in case of changes.

Topic-based Selection

When using topic-based selection, topics are identified by keywords (e.g. `stock quote`). Every topic is seen as an event service of its own to which consumers can subscribe. Topic-based systems are similar to group communication and event-notification systems.

Topic-based selection does have the disadvantage that it is rather static and that platform interoperability has to be enforced by limiting the keys to strings. Additionally its expressiveness is limited as a subscriber may have to subscribe to a topic although it is only interested in parts of this topic. This leads to unnecessary publication messages and therefore possibly inefficient use of bandwith.

The advantage of topic-based selection over channel-based selection is the better separation of publishers and consumers and the higher degree of flexibility.

Type-based Selection

Unlike the other approaches, the type-based selection mechanism checks commonalities not only in the content but also in the structure. Filtering is done according to the type and not according to the name. This can lead to a more natural representation of filtering. In object-oriented languages, type-based selection is implemented using inheritance.

Content-based Selection

Content-based selection is the most powerful and most flexible notification selection mechanism as events are classified according to their properties and not according to some predefined external criterion. This also allows for a finer granularity. With content-based selection, full decoupling of producers and consumers is possible:

Space decoupling The interacting parties do not need to know each other. Publishers do not hold references to subscribers, they even do not know how many subscribers are consuming their messages. Similarly, subscribers do not hold any reference to the publisher.

Time decoupling Publishers and subscribers do not have to be actively participating at the same time.

Synchronization decoupling The communication is asynchronous, clients are not blocked during the communication.

While content-based selection is the most powerful notification selection mechanism it is also the most complex one to implement.

2.1.5 Routing

In a publish/subscribe system, an efficient way to route messages from the producer to the consumers is necessary. A centralized notification service is undesirable for performance reasons. Instead, the functionality should be distributed over a set of cooperating event brokers [16] with each broker managing an exclusive subset of clients and propagating the messages. Therefore, messages have to be sent to these brokers and finally to the clients.

Messages could be flooded to all actors in the system but *flooding* leads to many unnecessary messages. Using the standard IP, routing is based on the destination address. This requires knowledge about the target and prohibits complete decoupling. As an alternative, content-based routing can be used.

When using *content-based routing*, messages are not routed with respect to their destination address but based on the actual content of the message. The content-based address of a subscriber is formed by the set of subscriptions it has issued. There exist three different versions of content-based routing (confer to [16]) which we shall now look at.

Simple Routing

Simple routing is the most basic content-based routing algorithm. All brokers have to know about all active subscriptions and each broker has an entry in its routing table for each active subscription. To achieve this, all new and cancelled subscriptions have to be broadcasted to all brokers.

Simple routing is not suitable for larger networks as numerous messages are sent unnecessarily.

Covering-based Routing

Covering-based routing is an improvement of the simple routing. Subscriptions are only forwarded selectively. A subscription that is already covered by a previous subscription is not forwarded anymore. A subscription F_1 covers another subscription F_2 iff $N(F_2) \subseteq N(F_1)$ where $N(F)$ is the set of matching notifications [15]. The covering is called *perfect* if equality holds, else it is *imperfect*.

Routing Using Advertisements And Subscriptions

*Advertisement*s are filters that are issued by publishers to inform about their intention to publish notifications matched by the issued advertisement. All

publications must be matched by a previously issued advertisement, that is the publication must have the same or fewer attributes than the advertisement. Additionally, the values for each attribute must be covered by the advertisement.

A *subscription* is a filter issued by the consumer to indicate interest to receive future publications. The notification set is the set of potential publications that would match the subscription.

The advertisements form an advertisement tree which is then used to route subscriptions. A subscription is only forwarded if it overlaps with an active advertisement. Similarly, the subscriptions form a subscription tree and a publication is only forwarded if it overlaps with an active subscription.

Using advertisements and subscriptions ensures that subscriptions and publications are not unnecessarily routed to brokers. This approach reduces the number of messages sent provided that advertisements are the least common message. If the number of publications is rather small in comparison to the number of advertisements and subscriptions, this approach is inefficient in terms of the number of messages.

2.2 Sub-2-Sub

Sub-2-Sub is an autonomous self-organizing peer-to-peer content-based publish/subscribe system that deploys an unstructured overlay network [20]. Sub-2-Sub supports value-based and interval-based subscriptions.

The dissemination of events to the subscribers is done by the cooperation of interested nodes themselves (autonomously), no relay servers or brokers are necessary. Nodes organize themselves in a structure in a completely decentralized manner (self-organizing).

2.2.1 System Description

Sub-2-Sub uses an epidemic algorithm to cluster subscribers. Each peer knows about some other peers. The peers it knows about comprise its view. This view is periodically updated with another peer. This updating is based on a proximity metric in the attribute space. Clustering of the subscribers is done according to the similarity in the subscriptions.

Subscribers use attribute values to define their interest. All attribute values have to be real numbers. The attribute values can either be discrete values (e.g. $A_1 = 5.25$) or ranges (e.g. $A_2 \in [3.18, 6.89]$). If multiple exact values or multiple non-continuous subranges (of the same attribute) are required, this has to be represented as multiple separate subscriptions. In case of multiple subscriptions, multiple virtual peers are run on a single physical node.

If two peers i and j note that their subscriptions intersect (i.e. their values overlap), they will maintain references to each other. If they discover a third peer k having overlapping subscriptions, they will associate with it. i and j will still keep references to each other but will reduce them to the difference between

their subspace and the subspace shared with k. Publishers join the network in the same way.

This procedure ensures that a publication will be delivered only to subscribers to it. However, it does not guarantee that all of them are informed. To achieve this, nodes self-organize into bidirectional rings. Additionally, nodes have three different types of links:

Random links Links to randomly selected peers in the overlay. Used to discover nodes and to keep the overlay connected in a single partition.

Overlapping-interest links Reflect the similarities between subscriptions, used to send published events to random other interested peers. Peers keep links to the closest nodes according to a given proximity metric. Sub-2-Sub defines proximity as the distance in the attribute space. Proximity is 0 if two nodes have overlapping interests and otherwise the Euclidean distance between the two subscriptions.

Ring links Used to build a ring of nodes. Peers periodically exchange the view with a neighbouring subscriber.

2.2.2 Spread Publications, Routing

If a publisher wants to publish an event/publication, it sends the message to a subscriber which will then disseminate it. Therefore the publisher only has to locate any matching subscriber and deliver the event to it.

Each node runs a dissemination algorithm that listens to incoming events and forwards them accordingly. On receiving an event, the algorithm checks if the event has already been seen, because previously seen events are ignored. If an event is new, it is delivered to the application and forwarded twice: to the two adjacent neighbours along the ring and to some (one or two) additional matching subscribers which are selected following a random link in the ring.

Client Arrival and Departure

As Sub-2-Sub is self-organizing, arriving and leaving clients do not represent a problem for the system. Details about the used procedure are not given in [20] nor was it possible to find any of them.

Fault Resiliency

The article presenting Sub-2-Sub does not specially mention fault resiliency. However the authors state that “epidemic protocols have proved to converge quickly and to produce failure-resistant overlays” [20].

2.2.3 Expected Network Traffic

The paper does not state anything about the network traffic and how the communication is actually done. The propagation speed (in measures of the number

of hops) increases with the number of matching subscribers in a logarithmic relation.

2.2.4 Evaluation

Advantages

Sub-2-Sub does have some interesting advantages:

- Accurate: all interested nodes - and nobody else - receive the event.
- Autonomous.
- Self-organized.
- Very scalable.

Disadvantages

Unfortunately, Sub-2-Sub does also have some serious disadvantages:

- Poor expressiveness of the subscription language: only real numbers can be indicated.
- Few information available, no implementation found.
- Multiple subscriptions by a node require multiple virtual peers on this node.

Conclusion

Sub-2-Sub does have some interesting features that would be useful for MMOGs. However, it is very new (2006) and there is very few documentation available. As a consequence it is impossible to really evaluate if Sub-2-Sub fulfils the requirements of MMOGs. Additionally, no implementation could be found.

Although it might fit the requirements, Sub-2-Sub does not seem to be the ideal candidate for a use in this project due to the lack of resources.

2.3 PADRES

PADRES (Publish/Subscribe Applied to Distributed Resource Scheduling)[5, 14] is a publish/subscribe system developed at the University of Toronto.

2.3.1 System Description

PADRES consists of a set of brokers to allow for distributed matching. Rather than using a centralized server, the clients connect to brokers that are connected by a peer-to-peer overlay network.

Routing

The basis of routing is built by a network of routers connected by an overlay network. Each broker knows its neighbours from an Overlay Routing Table (ORT). Messages are routed using advertisements and subscriptions (see to section 2.1.5 on page 7). Publications are assumed to be the most common message whereas advertisements are assumed to be the least common messages as they are flooded to all brokers.

Subscriptions (which may be issued by the client at any time) are routed according to the Subscription Routing Table (SRT). This SRT is created using the advertisements the broker receives and states to which broker subscriptions overlapping this advertisement have to be routed. The SRT is a list of `[advertisement, last hop]` tuples.

Publications are routed according to the Publication Routing Table (PRT) which is very similar to the Subscription Routing Table. Contrary to the SRT, the PRT is created using subscriptions. If a publication overlaps a subscription, it will be forwarded to the broker indicated in the Publication Routing Table.

Broker Architecture

Each broker has one input queue and multiple output queues, one per message destination. The matching engine uses the Java Expert System Shell (JESS) to maintain the SRT and PRT (both are represented as Rete trees). Publications are inserted into the matching engine as a fact. If a publication matches a subscription in the PRT, it is placed into the corresponding output queue (and forwarded to the broker/subscriber). If the message is a subscription, it is first routed according to the SRT and then inserted into the PRT as a rule.

Subscription Language

The subscription language of PADRES is notationally simple but still very powerful. Publications are pairs of {attribute, value}. An example for a publication is `([class, accident] [city, Toronto] [severity, 3] [victim, John Doe])`.

Subscriptions are very similar to publications and are represented as conjuncts of {attribute, operator, value}. `([class, eq, accident] [city, eq, Toronto] [severity, >, 2])` is an example for an atomic subscription.

Client Arrival

Arriving clients have to know a broker to which they connect. They then have to issue subscriptions to get publications.

Client Departure

If a client leaves, the client code automatically unsubscribes and unadvertises any state previously submitted to the broker it is connected to. The broker will then remove all subscriptions and advertisements belonging to this client. To complete, the client terminates the RMI connection with the broker.

Crash, kill or exit events of clients trigger the procedure to clean up. If the connection between the broker and the client is interrupted before, no clean-up happens. Currently, there is no periodic clean-up but according to the researchers at the University of Toronto plans are to add such a task.

Fault Resiliency

Padres provides fault resiliency in the default version. When detecting a broker or link failure, a recovery procedure is triggered which maintains the integrity of the broker network and updates the advertisement routing tables, the subscription routing tables and the publication routing tables.

Cycles in the network are used to accelerate the recovery of publication flows. Publications are automatically routed around failures. Therefore, routing of messages is reliable in Padres. Once a publisher/subscriber path is constructed, no publications are lost.

2.3.2 Special Features

Historic Data Access

In usual content-based publish/subscribe systems, subscribers can only subscribe to data in the future. In addition to that, Padres offers the possibility to subscribe to data published in the past. Historic data access is enabled through a database binding. Databases store publications when they are published.

If a broker receives a request for the historic data stored in the database, it re-publishes the information from the database. The data in the database cannot be queried. However, the database advertises the content stored in it and the subscription will be automatically routed to the correct database.

For the client, historic data access is transparent, it simply adds a time predicate to the subscription and receives data for the appropriate period.

Composite Subscriptions

Padres allows to create complex subscriptions by combining several subscriptions using basic relational operators such as `AND` and `OR`. Additionally, advanced features such as variables, sequences and repetitions are provided. Variables are used to join subscriptions. The sequence operator `;` defines the time sequence of two publications. As an example, the composite subscription $S_1; S_2$ means that a publication matching s_1 is followed by a publication matching S_2.

The subscription `Times(S, n, attr, v)` describes a repetition event pattern and means that publications matching S happen n times and attribute $attr$ increases by step v each loop.

A composite subscription is represented by a subscription tree. Internal nodes of the tree are logical operators, leaf nodes are primitive subscriptions. A composite subscription is routed as a unit until it reaches a joint point broker, that is the first broker at which the different data publishers contributing to satisfying the composite subscription are located in different directions in the overlay network.

As a composite subscription is forwarded into the network as far as possible before it is split, composite subscriptions reduce the network traffic.

2.3.3 Evaluation

In the following, some advantages and disadvantages of Padres are presented.

Advantages

Padres has many positive points:

- Powerful subscription language.
- Content-based.
- Reliable.
- Active project, improvements ongoing.
- Support by University of Toronto.

Disadvantages

Unfortunately, Padres has also some weak points:

- Brokers represent a new bottleneck.
- Uses RMI which represents a bottleneck.
- Advertisements flooded to all brokers.
- Not self-organizing, joining rather complicated.
- Point-to-point communication is not provided but both partners have to send a separate advertisement and subscription to simulate such point-to-point communication (see Section 3.3.1 on page 27 for a detailed description).

Conclusion

Padres is a very interesting and powerful content-based publish subscribe system. The expressiveness of its subscription language could be very useful for Mammoth, allowing to move some interest management to the brokers.

However, Padres does have three serious disadvantages: First, it is not self-organizing and thus the arrival of new clients is not managed automatically. Second, the brokers represent a bottleneck when scaling the system to a big number of clients (confer also [19]: "Existing solutions typically use a relatively small set of trusted computers as brokers, which may lead to scalability concerns for large Internet-scale workloads."). Third, Padres does not natively support direct messaging.

2.4 Pastry

Pastry [6] is "a scalable, distributed object location and routing substrate for wide-area peer-to-peer applications" [17]. Pastry is decentralized, scalable, and self-organizing. The original version of Pastry has been improved and simplified, considerably reducing the network overhead [10].

Each node in the Pastry network is assigned a randomly chosen unique numeric identifier, the `nodeId`. Each message to be routed has to be given to Pastry with a numeric key and will then be routed to the node with a `nodeId` that is numerically closest to the given key. When a message passes a node, the application is notified so that it can perform necessary computations.

To minimize the distance messages travel, Pastry uses a scalar proximity metric (e.g. the number of hops). `NodeId`s are randomly assigned which increases the probability that the neighbours of a node are in a different network.

As with other distributed hash table (DHT) systems, the strength of Pastry is its scalability and fault-tolerance.

2.4.1 System Description

All nodes in Pastry use the same software and "any computer that is connected to the Internet and runs the Pastry node software can act as a Pastry node, subject only to application-specific security policies." [17] Since the `nodeId`s are assigned randomly when a node joins the system, nodes with adjacent `nodeId`s are usually (with a high probability) located in different networks, reducing the impact of local network problems.

Routing

`NodeId`s and keys are represented by a sequence of digits with base 2^b where b is a configuration parameter (typical value of 4). Messages are routed to the node whose `nodeId` is numerically closest to the given key.

Each of the N nodes in the Pastry network has to maintain two state tables:

Routing table The routing table has $\lceil \log_{2^b} N \rceil$ rows, each of which has $2^b - 1$ entries referring to a node (by indicating the IP address) with a `nodeId` sharing the first n digits with the present node's `nodeId` but where the $n+1$th digit is different from the one in the present node's `nodeId`. When choosing the node, nodes close to the present node are preferred. Proximity is determined according to the proximity metric.

Leaf set The leaf set L is the set of nodes with the $|L|/2$ numerically closest larger `nodeId`s, and the $|L|/2$ numerically closest smaller `nodeId`s. $|L|$ is a configuration parameter, with typical value of 16 or 32.

Note that the neighbourhood set defined in the original version of Pastry has been dropped in the new version.

On *receiving a message*, the node checks if the key of the message is covered by the leaf set L, i.e. if it falls within the range of `nodeId`s covered by the leaf set L. In this case, the message is forwarded to the node in the leaf set whose `nodeId` is closest to the key (i.e. which is the destination node of the message, may be the present node).

If the destination node is not in the leaf set L, the routing table is used to forward the message to a node which is chosen based on the length of a prefix that the `nodeId` shares with the key. This prefix shared by the next node and the key must be at least one digit ($= b$ bits) longer than the prefix shared by the current `nodeId` and the key. If there is no such node, the message is forwarded to a node sharing with the key a prefix of the same length as the current `nodeId` but that is numerically closer to the key.

Proximity is defined as a scalar proximity metric (number of IP routing hops or geographic distance).

Routing Maintenance

As the join protocol (cf. 2.4.1) as well as the repair mechanisms do not guarantee to produce the routing table entries closest to the local node, a special route maintenance task is triggered periodically by each node. The node randomly selects an entry in each row of its routing table. It then requests the corresponding routing table row from the node associated to this entry to compare the entries. If the entries differ, the node selects the closer one (by probing) and installs it in its own routing table.

Node Arrival and Departure

As Pastry is completely decentralized and self-organizing it automatically adapts to the arrival, departure or failure of a node.

Node arrival An arriving node with `nodeId` X has to contact an existing node A. A routes a message using X as key. This message will be delivered to Z, the node with the `nodeId` numerically closest to X. The joining node receives the nth row of the routing table from each node encountered on the

path from A to X whose `nodeId` matches X in the first $n-1$ digits. The leaf set of X is set to the leaf set of Z.

As soon as X has completed its routing table, it sends the nth row of it to each node appearing as an entry in this nth row. The purpose of this is both to announce its arrival and to propagate information about other nodes that have joined previously. The nodes receiving a row will then check if one of the entries in the row is nearer than the corresponding entry in its own routing table. If this is the case, the routing table will be updated.

Node departure A node detects that another node has failed or left when it tries to contact this node and does not get a response. If the failed node L_i was in the lower half of the leaf set of the surviving node L_0 (i.e. $-\lfloor |L|/2 \rfloor < i < 0$), L_0 will request the leaf set from $L_{-\lfloor |L|/2 \rfloor}$ and extract the nodes from it that are not in the leaf set of L_0. From these nodes it will then choose one to insert into its leaf set after having verified (by contacting it) that the new node is alive.

If a node appearing in the routing table of another node fails, the routing entry must be replaced. If Pastry discovers that a node has failed (because contacting this node to route a message failed), the message is routed to another node with a numerically closer `nodeId` but with the same prefix (meaning that this node is in the same row of the routing table as the failed node).

Malicious Nodes

If a node really fails and does not react anymore, the problem can be solved in the way described above. If the node continues to be responsive but has started to behave incorrectly or maliciously, Pastry does not realize that. To change that, routing would have to be adapted in that the node to which the message is forwarded would be selected randomly from the nodes satisfying the criteria (sharing a longer prefix with the destination (key) or having the same prefix but with nodeId numerically closer than the current one).

Network Partitions

If certain hosts are unreachable for some of the hosts but not for others, multiple network partitions will be created as Pastry is self-organized. This partition may persist even though the two partitions could be reconnected, for example because the network has been repaired. To overcome this problem, Pastry can use IP multicast to periodically perform an expanding ring multicast search for other nodes in the neighbourhood. Using this method, isolated partitions will be discovered and reintegrated.

2.4.2 Performance

The expected number of routing steps is $\lceil \log_{2^b} N \rceil$ but the performance degrades gradually with the number of recent node failures. Messages are always delivered unless $\lfloor |L|/2 \rfloor$ nodes with consecutive `nodeId`s fail at the same time.

2.4.3 Evaluation

Advantages

Pastry has several important advantages for a use in MMOGs:

- Self-organizing and therefore failure resistant.
- Peer-to-peer, scalable.
- All nodes are equal, no special infrastructure (servers, brokers etc).
- Protocol to locate an appropriate contact node for nodes wanting to join is provided. However, one node must be known to be able to join.
- Few messages sent.
- Software available.

Disadvantages

As every system, Pastry also has some inconvenient properties:

- Proximity metric does not always work as expected (if triangle equality is not satisfied).
- Join protocol and repair mechanism do not guarantee to find the closest node.
- Peer-to-peer might abet cheating.
- No reliability guarantees.

Conclusion

Pastry seems to be a very interesting substrate that could be extremely useful for MMOGs. Many key features of Pastry, namely the fact that it is distributed and self-organizing, match the requirements of MMOGs. However, Pastry does not provide any publish/subscribe functionalities and is not reliable.

2.5 SCRIBE

Scribe [8, 11] is a "large-scale event notification infrastructure for topic-based publish-subscribe applications" [18]. Scribe is based on Pastry (see Section 2.4 on page 14).

2.5.1 System Description

The aim of Scribe is to provide an efficient multicast on the application layer. Based on Pastry and thus using the peer-to-peer-technology with good partitioning and distributing forwarding load over all nodes, Scribe is capable of scaling to a large number of subscribers, publishers and topics. Basic Scribe does not provide reliability guarantees nor ordered delivery for a topic, it provides only best-effort dissemination. Stronger reliability guarantees have to be built on top of Scribe.

Scribe provides the following API (source of API description: [18]):

create(topicId,commonName) creates a topic with topicId and the name commonName.

subscribe(topicId, client), subscribe(topicId, client, scribeContent) causes the given node to subscribe to the topic with topicId. All subsequently received events for that topic are passed to the specified node.

unsubscribe(topicId, client) causes the local node to unsubscribe from the topic with topicId.

publish(topicId, scribeContent) causes the message 'scribeContent' to be published in the topic with topicId.

Other than specified in [18], Scribe does not integrate credentials into the API. However, credentials can easily be added to the `scribeContent` which is the message (object) to be sent. It is possible to add such an object to the subscriptions. Verification of these credentials can be done in a `scribePolicy`.

As Scribe is fully decentralized and based on peer-to-peer, "each node can act as a publisher, a root of a multicast tree, a subscriber to a topic, a node within a multicast tree, and any sensible combination of the above." [18]

When using Scribe, each node in the network of Pastry nodes has to run the Scribe application software. This software has to provide the forward and deliver methods. The forward method is used when a message is routed through a node whereas the deliver method is called when the message has arrived at the destination node. Scribe supports four types of messages: subscribe, create, unsubscribe and publish.

Pastry/Scribe support two implementations for the message format. Originally, Java Serialization was used for all messages. As of version 2.0 of FreePastry Raw Serialization, which is more efficient but requires additional work, is also supported.

Topic Management

Each topic in Scribe has a unique `topicId` (corresponding to the key in Pastry). For each topic there is a rendez-vous point. There are two possibilities to choose the rendez-vous point: either the node with `nodeId` closest to the `topicId` or

the creator of a topic. To make the creator of a group be the rendez-vous point, a `nodeId` can be the hash of the textual name of the node and the `topicId` can be the concatenation of the `nodeId` of the creator and the hash of the textual name of the topic.

To create a topic, a special message with the `topicId` as key is routed by Pastry. By definition of Pastry, the message is sent to the node with `nodeId` numerically closest to the `topicId`. This node (its delivery method) then adds the topic to the list of topics it already knows about.

Routing

Scribe uses multicast trees for routing. The root of this tree is at the rendez-vous point of each topic (i.e. the node with `nodeId` closest to the `topicId`). "The tree is formed by joining the Pastry routes from each subscriber to the rendez-vous point" [18]. All nodes in this tree are so-called forwarders for this topic (not necessarily subscribers). Each of them has a children table, that is a routing table containing all its children in the multicast tree.

Subscriptions A node that wants to subscribe to a topic has to issue a subscribe message with the `topicId` as key. The message will be routed by Pastry towards the rendez-vous point of the topic. At each node the message passes, the forward method is invoked which will then check if the node is already a forwarder. If this is the case, the subscribing node is registered as a child (and the subscription message is stopped). If the current node is not yet a forwarder for the topic, it adds the subscribing node as a child, creates an entry for the topic and issues a new subscription to this topic.

A well described example can be found on page 4 of [18] (starting at "Figure 4 illustrates...").

Unsubscriptions To unsubscribe, a node marks the topic as no longer required. If the node does not have any children for this topic, that is the children table is empty, an unsubscribe message is sent to its parent in the multicast tree. The parent will then remove the child and verify if the unsubscribing child was its last child; if so, the unsubscribe message is forwarded to its parent which will do the same handling etc. If there are any other children, the message is not forwarded.

Note that nodes do not know about their parent until they receive a message (event, publication) from the parent. If the parent is not already known, the unsubscription is delayed transparently until the parent is known which is the case as soon as the first event is received.

Publish, event dissemination If a publisher knows the IP address of the rendez-vous point, the publish message can be sent directly to the node. Otherwise, the message is routed using Pastry and the rendez-vous point is then asked to return its IP address to the publisher which will cache it (as an optimization).

If the rendez-vous point has changed (e.g. arrival of a new node), the publisher sends the message to the wrong node. This node will then forward it to the new rendez-vous point and ask it to communicate the IP address to the publisher.

Reliability

The basic version of Scribe does not provide reliability guarantees but provides only best-effort delivery of events. The delivery of events is only reliable and ordered if the TCP connections between the nodes in the multicast tree are not broken. Scribe may fail to deliver events or may deliver them out of order if some nodes in the multicast tree fail.

Provided Reliability Failures in the multicast tree are detected by having each leaf node in the tree send a heartbeat message (subscription) to its parent. If a child discovers that its parent is faulty because the subscription message cannot be routed quickly, it uses Pastry to route it. This routing will repair the multicast tree. With the default parameters, detecting failure in FreePastry takes about 30 seconds after trying to send a node a message.

In Pastry, children periodically (e.g. every 3 minutes) send a subscription to their parent to confirm their interest in the topic. Otherwise, they will be removed from the children table.

2.5.2 Evaluation

Scribe seems to fulfil many requirements of MMOGs. We shall now look at some advantages and disadvantages as well as test the implementation using the tutorial[1].

Advantages

Being based on Pastry, Scribe does have the same advantages as Pastry (confer Section 2.4.3) but the following points shall be mentioned specially:

- The underlying layer (Pastry) allows to route messages directly to a node. Therefore, the combination of Scribe and Pastry offers both direct messaging and publish/subscribe.
- Security features can be added.
- Contact established with researcher at Max Planck Institute for Software Systems, Germany.
- Active project, research ongoing with special focus on increasing the performance (reduce bandwidth/cpu/memory) and implementing NAT-support.

Disadvantages

Unfortunately, Scribe has some undesired characteristics:

- Topic-based selection mechanism is less powerful than content-based mechanism and the set of available topics has to be defined and known to clients.
- Only best-effort reliability guarantees, extension necessary for stronger reliability.
- Very large numbers of subscribers to a topic can overload the corresponding rendez-vous point.
- Security features are not provided by default.
- Support for NATted nodes is currently broken, clients behind a firewall cannot connect to the network.

Practical Test

The implementation of Scribe/Pastry has been tested several times, following the guidelines in the tutorial[1]. There were no problems encountered during this testing, all tests worked well. Scribe really seems to be ready for use.

Conclusion

Scribe seems to perfectly match the requirements of MMOGs as it is self-organizing, distributed using peer-to-peer and allows for both direct messaging and publishing. The disadvantage is the lack of expressiveness of the language as it is topic-based. However, this should not be a problem regarding Mammoth as Mammoth currently uses channels to communicate. Topic-based publish subscribe is slightly more expressive than channel-based publish subscribe. Therefore, the change from a channel-based to a topic-based system should not present a problem.

2.6 Evaluation Summary

The four systems described above are all powerful systems that are very interesting and might all be helpful in a way for MMOGs. All of them do have many advantages but unfortunately also some disadvantages. We will now choose two systems to be discussed together with the requirements of MMOGs such as Mammoth in a detailed evaluation given in Chapter 3.

Sub-2-Sub does have some interesting points, namely that it is a content-based publish/subscribe that is extremely scalable as it is self-organizing and uses peer-to-peer. However, there is very few information available about Sub-2-Sub and the authors declare themselves that there is a lot of work to be done

to make it appropriate for real use. Therefore, Sub-2-Sub will not be considered for use in in MMOGs.

PADRES is a very interesting and very powerful distributed content-based publish/subscribe system. Unfortunately, it does not provide a native way for direct messaging. Additionally, the brokers and RMI represent a possible bottleneck. A big advantage of PADRES is that it has been developed at the University of Toronto and that its researchers would be willing to adapt Padres to better comply with the requirements of MMOGs. Therefore, PADRES will be considered in the detailed evaluation.

The big strength of Pastry is its scalability which is due to the fact that it is completely distributed and peer-to-peer. However, Pastry is not a publish/subscribe system but only an object location and routing substrate. Publishing is not provided, only one-to-one communication is possible. To be really useful, an application has to be built on top of Pastry. Therefore, Pastry will not be considered in the detailed evaluation.

Scribe is a topic-based publish/subscribe system based on Pastry. It thus has the advantage of being scalable and the ability to use functionalities provided by Pastry which is of special interest for the direct messaging. Unfortunately, Scribe is not content-based which limits its expressiveness. Nevertheless it shall be considered in the more detailed evaluation as the overall impression is very good.

Chapter 3

Requirements of MMOGs and Solutions

The aim of this chapter is to discuss the requirements and possible solutions for the network layer of MMOGs using Mammoth [4] as an example. First, a short introduction to Mammoth is given. Second, the necessary communication within Mammoth is described on a relatively high level of abstraction. Third, some concrete implementation problems are presented together with possible solutions with Padres and Scribe. Last, one of the systems is chosen to be actually implemented.

3.1 Mammoth

Mammoth is a "massively multiplayer game research framework. It's goal is to provide an environment for experimentation in areas such as distributed systems, fault tolerance, databases, modeling and simulation, artificial intelligence and aspect-orientation. The Mammoth Project was launched in the summer of 2005 as a collaborative project among several McGill professors and a number of graduate students." [4] Mammoth has been developed in Java.

Mammoth is a role-playing 'game' but unlike a real game it does not have an actual goal. The range of possible actions is very limited. However, the possible actions are equivalent to more complicated actions in commercial games in terms of the underlying treatment. The main activities are walking around and picking up or dropping items such as tomatoes and flowers. Furthermore, different players may communicate using a chat box. In addition to the players and the items there are obstacles such as trees or walls in the world of Mammoth. These obstacles prevent players from moving in a straight line. Obstacles are defined statically at creation time and cannot be moved around or modified in another way.

Currently, the environment of Mammoth is two-dimensional as shown in Figure 3.1 and the player sees the world from above. However, a three-dimensional

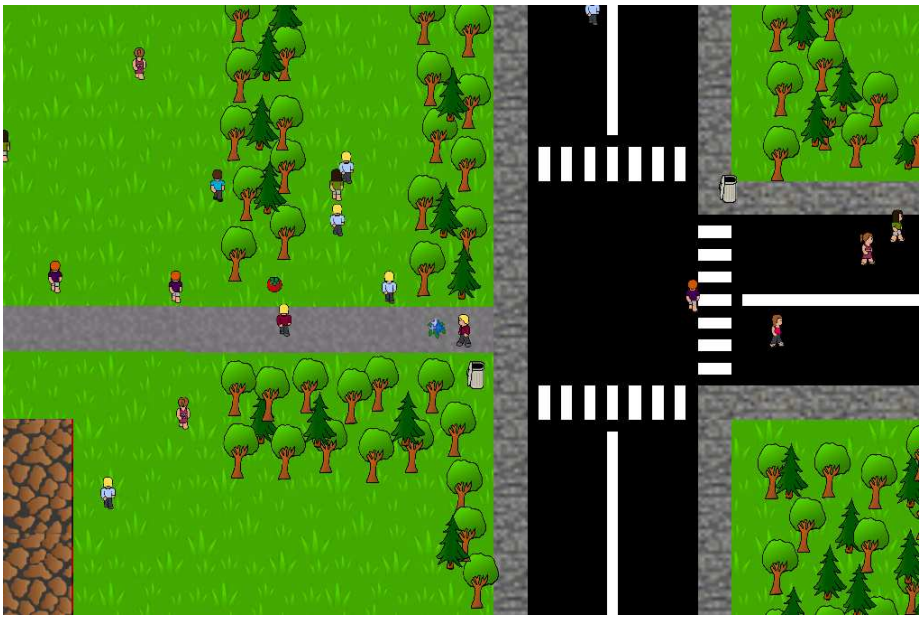

Figure 3.1: The current 2D view in Mammoth.

Figure 3.2: The beta version of the 3D view of Mammoth.

version (see Figure 3.2) is being developed as a separate project branch.

Mammoth uses interest management algorithms to determine the replicas that have to be distributed by a replication engine (see Chapter 5.2 of [9] for a detailed explanation of the replication engine). With this structure, a central server does the interest management for all objects. Each user of Mammoth gets duplicas of the objects in the world. The server is the master of all these duplicas, that is to modify the state of an object a duplica has to ask the master for permission. If the master grants the permit for the change, it has to inform all duplicas about the change of the state to keep the system consistent. In Mammoth, this is done by sending a message to the duplicas.

The software architecture of Mammoth is very modular (see Chapter 5.1 of [9]) so that components can be replaced easily. One of the modules is the network layer, which is the most important component for this project. The network layer is not specific to the game. The default network layer used by Mammoth is a simple client-server architecture in which the server knows about all clients and communicates directly to all connected clients through a TCP connection. To inform a set of clients about state changes, the master sends a message to each client. This architecture introduces a serious bottleneck which should be disposed of in the frame of this project.

3.2 Requirements

A system for the network layer of MMOGs such as Mammoth should allow for publishing as well as direct messaging (see also [12]).

3.2.1 Scalability

The network layer should support massive scalability, ideally it should be able to scope with the traffic caused by 3000 simultaneously connected players. In more detail this implies the following requirements (defined in [12]):

- Create up to 5000 channels on a node. A publisher should be able to publish on 5000 different channels.
- Client subscribes to up to 5000 channels.
- Up to 1000 clients simultaneously subscribed to a channel.

3.2.2 Message importance/subscription quality

As described above, Mammoth requires a high performance in terms of the quantity of messages sent. Additionally, it also needs a high quality of the messaging service.

Priority Possibility to add priority to messages to influence the order in which the messages are handled at the destination.

Reliability Mammoth does require reliability in the sense that all direct messages must be delivered exactly once. Ideally, there would be two possibilities: reliable sending for important messages (e.g. item grabbed) and unreliable sending for less important messages like information about movements.

Ordering For the current version of Mammoth, strict delivery order is not of key importance for different objects but it is important for updates of the same object. However, this must be implemented on the level of Mammoth and is thus not within the scope of this thesis.

3.2.3 Direct Messaging

Direct messaging between two clients is necessary for Mammoth to remotely invoke a method and send the reply back to the invoker. Ideally, the following features are supported by a system:

Synchronous Ideally, synchronous communication would be provided by the new network layer. However, publish/subscribe systems are asynchronous by definition and thus native synchronous communication is impossible with publish/subscribe systems but has to be simulated.

Asynchronous Network latency should not block the game for an unnecessary long time. Blocking the game until a reply message arrives is undesirable. Therefore, asynchronous communication should be provided.

Reply A receiver should be able to send an answer to an incoming message.

3.2.4 Interface

The network layer should provide at least the following API to higher levels:

advertise(...) Publish/subscribe. Used by publishers to advertise future publications. Only necessary with routing based on advertisements.

unadvertise(...) Publish/subscribe. Used by publishers to remove advertisements. Only necessary with routing based on advertisements.

subscribe(...) Publish/subscribe. Subscribe to publications. Separate interfaces have to be provided so that a client can subscribe not only itself but also another client.

unsubscribe(...) Publish/subscribe. Unsubscribe from publications. The capability to unsubscribe other clients from topics is required.

publish(...) Publish/subscribe. Issue a publication.

send(destination, msg) Direct messaging. Send the message `msg` to `destination`.

3.2.5 Arrival of Clients

Arriving clients should be able to easily integrate the system at any time.

3.3 Problems & Solutions with PADRES and SCRIBE

In this section, the problems to be solved and their possible solutions with PADRES [14, 5] (confer Section 2.3 on page 10), a content-based publish/subscribe system, and Scribe (see Section 2.5 on page 17), a topic-based publish/subscribe system based on the DHT Pastry [17, 10, 6] (see Section 2.4 on page 14), are described.

3.3.1 Communication Duplica ↔ Master

Duplica have to be able to send a request to their master which must then be able to send a reply. Example: `Player 1` wants to take the apple `a1` which is represented in the players duplication space by the duplica `D2`. `D2` then has to ask the master `M` whether the apple `a1` can be taken away. If this is the case, the master has to tell all duplicas of apple **a1** that apple **a1** has been grabbed. If the apple **a1** cannot be grabbed, the master has to send an appropriate answer to **D2** (and only to **D2**).

Problem: Invocation of Method on Master

The duplicas have to be able to contact the master `M`, e.g. to invoke a method on this master.

PADRES To invoke a method on the master, a message has to be sent to the master. The duplicas all send an advertisement to announce their intention to send such messages. The advertisement should look like `adv [class,eq,invocation],[item,eq,a1]`. The master then subscribes to invocations of item **a1** by sending a subscription `sub [class,eq,invocation],[item,eq,a1]`. To invoke a method, the duplica will now create a publication similar to `pub [class,eq,invocation],[item,eq,a1]`.

The following *requirements* have to be fulfilled for this method to work efficiently:

- Advertisements are only forwarded if it is necessary, i.e. iff no other advertisement covering the new one has been forwarded before. This avoids unnecessary advertisements if several duplica are connected to the same broker.
- It should be possible to integrate additional information that has not been advertised to a publication.

SCRIBE To invoke a method on the master, a message has to be sent to the master. Using the underlying Pastry, a message can be sent directly to a specific node (in this case the master). This method only works if duplicas know the `nodeId` of their master.

Problem: Direct Reply by Master

After the invocation, the master has to send a reply to the invoking duplica and only to this duplica. This direct messaging is not directly within the cope of publish/subscribe and we have to add this functionality or to simulate it.

PADRES

Using Advertisements/Subscriptions The solution using pure content-based publish/subscribe is to let the master send an advertisement for each duplica to announce that it is going to 'publish' something about/to this duplica. Each duplica would then send a corresponding subscription (each duplica subscribes only to publications targeted at itself!). Now, the master can simply publish a publication that will be forwarded only to the intended duplica.

This solution follows the publish/subscribe design but leads to a considerable network traffic. Assuming that there are about 1000 items in the virtual world of Mammoth and that 3000 players are connected, each player has to receive a duplica of each of these items. There are $3000*1000 = 3000000$ duplicas created. As each of these duplicas must be able to receive a reply (direct message) from its master, this would mean that 3000000 advertisements (one advertisement by duplica) are broadcasted by the master. Furthermore, 3000000 subscriptions would be sent by the duplicates just to be able to send answers from the master to the duplicas. As these advertisements are all different they cannot be avoided. However, these advertisements are not sent all at the same time, for each client connecting there are two advertisements and two subscriptions (one of each type for each of the master and duplica) sent.

This procedure assumes that the master knows (the IDs of) the duplicas it manages to be able to integrate the ID into the advertisement announcing the intention to send something to such a duplica.

A possibility to reduce the amount of advertisement and subscription messages could be to aggregate items together in groups (of a certain size or by their class) and then to have the clients do some additional filtering.

Another way to reduce the number of advertisements and subscriptions by moving some filtering to the client is to send rather unspecific advertisements. Instead of sending several very similar advertisements with only a single attribute changing, a master could send a single one and not specify the diverging attribute and add extra information as a serialized object. This would allow the client to decide whether the message is important or not. Note that this means that more traffic for publications is generated on the network.

If a master moves so that it is connected to another broker all routing tables have to be updated. Therefore the master should not be moved to another broker.

Reversed Routing The idea is to hack the routing of Padres so that the path described by the advertisements sent by the duplica (and normally used to route subscriptions from the master) is used to route a message from the publisher to the client. Publications for direct messaging would follow the advertisements (coming from the destination) instead of the subscriptions.

This procedure would save messages but means that the system has to be hacked and that there are two types of publications. A separate routing mechanism would have to be implemented.

SCRIBE Like the message to invoke a method, with Scribe the reply could also be sent using the functionalities of the underlying Pastry.

Problem: Published Reply by Master

A master must be able to communicate state changes to all interested items and duplicas.

PADRES To publish state changes, the master issues an advertisement announcing that it is going to publish something on this item. Clients interested in such messages create a subscription and will thus get publications issued by the master.

All duplicas have to subscribe to their master.

SCRIBE For each item, a topic is created to which interested items and duplicas can then subscribe to get information about state changes. Each node in the network can publish messages to this topic.

3.3.2 Information about Moving Players

A player is a replicated object. Thus it simply informs its master about moves. Informing the master is done using method invocation (see Section 3.3.1). It is then up to the master to inform all duplicas of the moving player that this player has moved (using the method described in Section 3.3.1).

3.3.3 Send Serialized Data

In Mammoth, the capability to send serialized data is required. It should not necessarily influence the routing.

PADRES In content-based publish/subscribe systems that use advertisements, all publications have to be advertised in advance and the entire content of a publication is checked against the subscriptions for routing. Generally it is not possible to send additional information. However, publication messages in Padres carry a serializable payload which is normally set to null and does not get involved in matching. Currently it is used for the monitoring in the network where brokers attach properties and performance metrics of the broker. It would be possible to use this object to add any other information that is neither used for routing nor advertised.

SCRIBE As Scribe is a topic-based publish/subscribe system the content of a message is not used to route the publication. The message already is a serialized object that is defined by the application using Scribe and that can contain any information.

3.3.4 Arrival of Clients

PADRES Arriving clients do have to connect to a broker. The list of brokers is static and defined in the properties. When a client wants to connect to the network, it randomly chooses a broker (in the static list) and connects to it.

SCRIBE Scribe/Pastry is self-organizing and arrival of clients is handled automatically. However, a list of join nodes has to be indicated. Clients willing to join will then iterate through this list of nodes until they find a working one.

3.3.5 Possible Improvements/Extensions

The following improvements/extensions would be possible (later):

PADRES

- Move some game logic to the broker: in case of competitive invocations the broker does not forward any other publications (that is invocations) for the same item if one has been sent before and the answer has not yet arrived. Instead it directly sends a message of refusal to the requesting client.
- Arriving clients should be automatically forwarded to a broker with free capacities, for example using load balancing. Problem: load balancing does introduce some conflicts with other features of Padres. According to Alex Cheung (University of Toronto) 'resolving this issue is on the long term agenda.'

SCRIBE

- Pastry does not provide any reliability guarantees. However, MMOGs require reliability. Thus, reliability would have to be added on top of Pastry.
- Pastry uses peer-to-peer and does not protect against malicious nodes. As described in Section 2.4.1 on page 16, random selection of the node to which the message is forwarded could be implemented to reduce the impact of malicious nodes.

3.4 Comparison

Table 3.4 on page 32 shows an overview of the different features required or desired for Mammoth and indicates for both Padres and Scribe if they are provided.

Table 3.1: Comparison of Padres and Scribe regarding Mammoth's needs.

Criterion	**PADRES**	**SCRIBE**
Reliability	default	possible to add
Scalability	brokers bottleneck, uses RMI	designed for scalability
Priority	ok, can be used for filtering	added to serialized object, no filtering
Send additional, unfiltered data	yes, using payload	default, no filtering
Synchronous	no	no
Asynchronous	yes, completely	yes
Direct messaging	no, to be simulated	yes, native (with Pastry)
Arrival of client	no automatic handling	automatic handling, self-organizing
Departure of client	limited automatic handling	automatic handling, self-organizing
Decouple clients	yes, completely	yes, topics to be defined
Send serialized objects	using payload	default message
Expressiveness	very good	average
Fault resiliency	average	good
Network traffic	Advertisements flooded	optimized
Peer-to-Peer	no	yes
Distributed	distributed matching	fully distributed
Support	perfect	help from developer and mailinglist
Availability of code	closed source due to the use of Jess, to be changed	BSD-like

3.5 Final Decision about Middleware

Scribe and Padres are both very interesting publish/subscribe systems and do have some advantages for Mammoth. While Padres is brilliant at the level of the subscription language with its high level of expressiveness, the strength of Scribe lies in the scalability and the fact that it is self-organizing.

Direct messaging, which is of key importance for MMOGs, has to be simulated using many messages in Padres. Scribe, in contrast, provides native direct messaging by means of the underlying Pastry. In terms of scalability, Scribe is better than Padres as the brokers used in Padres represent a bottleneck.

The main disadvantages of Scribe are the lacking reliability guarantees and its relatively limited expressiveness as it is topic-based. By definition topic-based publish/subscribe systems are less powerful in terms of the expressiveness than are content-based publish/subscribe systems. However, the precise definition of the requirements of the Mammoth network layer has shown that a topic-based publish/subscribe system is sufficiently powerful for Mammoth.

All in all, Scribe seems to be a complete system that perfectly fits the requirements of Mammoth and that can be easily integrated into Mammoth without having to modify it, whereas Padres needs some additional development and some modifications to be really suitable for Mammoth and MMOGs in general. Therefore, Scribe/Pastry (or rather its open source version FreePastry 2.1) will be used to implement the default version of Postina, the new API for the network layer of MMOGs.

Chapter 4

Postina

The aim of this Chapter is to introduce Postina. First, a general description explaining the capabilities of Postina is given. Second, the API provided by Postina is described in some detail. Third, the internal functioning of the default implementation of Postina with Pastry/Scribe is described. Fourth, a user guide explains how Postina can be used in other applications. Last, the unit testing of Postina is discussed.

4.1 Description

In this section, Postina, the new API for network layers for MMOGs, shall be presented in a comprehensible, generally understandable way.

4.1.1 Purpose

Postina is an API designed for the special requirements of network layers in massively multiplayer online games. That is, it provides interfaces for both direct messaging as well as publish/subscribe functionalities for the communication and messaging necessary in such games. The API has also been implemented in Java using Pastry and Scribe, resulting in a framework that is ready to use and combines the capabilities of Pastry and Scribe, adds some missing functionalities and offers an interface specially adapted to MMOGs.

The concrete realization of Postina is an application built on top of Pastry and Scribe (see Sections 2.4 and 2.5 for a description of Pastry and Scribe, respectively), which were both built with a special focus on scalability. Based on Pastry, the provided implementation of Postina is a peer-to-peer-application in which each client (node) is equal to all others. All clients build a network and can communicate with each other. Each application executed on a separate computer represents a client in the P2P-network.

Although the provided implementation of Postina uses Pastry and Scribe, this is not the only possibility. Postina is designed in a way that Pastry and

Scribe can be replaced without having to change the application that uses Postina.

4.1.2 Naming

Postina is an API for the network layer which takes care of the messaging in MMOGs. The name chosen for Postina reflects this main feature: "la postina" is the Romansh[1] word for a "female mailcarrier". The proper pronunciation[2] is pɔʃˈtɪmᵊ.

4.1.3 Features

Postina offers several features that are of key importance for MMOGs for communication between different players. First, methods for connecting and disconnecting are provided. Second, a message can be sent directly to a specified user in an unreliable or alternatively in a reliable (but less efficient) way. Third, a client can publish information to a topic and all clients subscribed to this topic are then informed about the publication. Using this publish/subscribe functionality, multicast communication, that is communication to a group of clients, can be done easily and in an efficient way. To receive a publication, a client first has to issue a subscription to the corresponding topic. Please see Section 2.1.4 on page 6 for a more detailed explanation of topic based publish/subscribe. Fourth, a client can require another client to subscribe to a specified topic. This allows to delegate the management of subscriptions to a special client. This can be used to let a client do interest management for other clients, that is allow a client to decide in which topics another client is interested.

Any `Serializable` can be sent as a message in Postina. For the delivery of messages, Postina offers two possibilities. The application using Postina may register as a listener which will then be informed about incoming messages, or messages can be put into a queue where the application may then (periodically) retrieve the messages from. It is not possible to use both methods simultaneously as registering a listener deactivates the queue to cleanly separate the two approaches. However, an additional queue can be emulated easily in the overlying application when listeners are used.

4.1.4 Special Features

As mentioned above, Postina provides some special features in addition to standard publish/subscribe functionality and direct messaging. These features are useful in massively multiplayer games and shall now be discussed in some more detail.

[1] Romansh is a Rhaeto-Romance language and the fourth national language in Switzerland. Please see `http://en.wikipedia.org/wiki/Romansh` for more information.

[2] Note that the Italian word "la postìna" has the same meaning but is pronounced differently: posˈtiːna.

Remote Calls

Currently, Postina provides a special feature that allows to issue subscriptions and unsubscriptions for other clients. With standard publish/subscribe systems this is not possible.

Reliability

Massively multiplayer games require reliable direct messaging between two clients (see 3.2.2 on page 25). Postina guarantees that each direct message is delivered at maximum once. Additionally, all messages are guaranteed to be delivered under normal circumstances. However, it is not absolutely guaranteed that the message is delivered as the maximum number of attempts is limited to avoid starvation. If a message has not been confirmed after this configurable number of attempts, Postina gives up and does not try again to send the message. With a reasonable configuration, the direct messaging can be considered reliable but a badly chosen parameter can break the reliability.

Postina does not provide any reliability guarantees for published messages. Reliability is provided for direct messages only. Furthermore, Postina does not guarantee ordered delivery. Each message is delivered once but the order in which the messages arrive may be different from the order in which they were sent.

Detection of Dead Clients

Postina provides a mechanism to detect dead clients, that is a client that has left the network without disconnecting properly. This can happen when the client crashes. In such cases it is important that this event is handled correctly in order to prevent the server from constantly sending messages to a dead client.

The detection of dead clients is done separately by each node in the network. Once a client has been declared dead, messages to it will not be sent anymore and a registered listener is informed about the fact that the client has left.

Broadcasting

Postina allows a message to be broadcast to all clients in the network.

4.2 API

This section gives an overview of the Application Programming Interface (API) provided by Postina. Developers requiring a more detailed description of the API are invited to look at the JavaDoc available on Postina's website (`http://postina.zindel.org`).

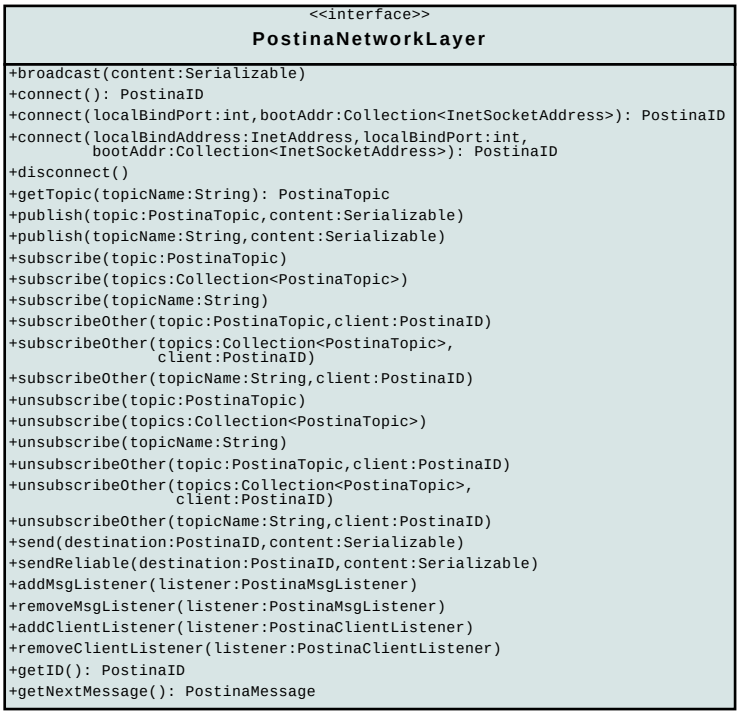

Figure 4.1: The Interface `PostinaNetworkLayer`.

4.2.1 PostinaNetworkLayer

The interface `PostinaNetworkLayer` which is also shown in Figure 4.1 on page 37 is the main interface of Postina and builds its core. Following, the main methods are described shortly:

broadcast The method `broadcast` allows a message to be sent to all clients in the network.

connect The method `connect` has to be called at first, before any other operation is possible. The method exists in three different variations: The version without any parameters reads all properties from the property-files as explained in Section 4.4.4 on page 49. The two other versions allow to specify the parameters used when establishing the connection.

disconnect The call of `disconnect` properly disconnects the client from the network and closes all connections.

getTopic The method `getTopic` allows the application to get the `PostinaTopic` for the specified topic.

publish The method `publish` provides the possibility to release a publication to a specified topic. For convenience, a method accepting a string name of a topic is provided.

subscribe The method `subscribe` is used to subscribe the client to a topic. The client will then receive all future publications to the specified topic. For performance reasons the method using a `Collection` of `PostinaTopic` should be used whenever a client has to register to multiple topics at the same time. For convenience, an additional method accepting a string name of a topic is provided.

subscribeOther The method `subscribeOther` allows to call `subscribe` (see above) on the client whose `PostinaID` is given.

unsubscribe The method `unsubscribe` is the counterpart of `subscribe`. Calling it unsubscribes the client from a topic and it will not receive any publications to this topic in the future. As for `subscribe`, the variation using a `Collection` of `PostinaTopic` should be used whenever a client has to register to multiple topics at the same time. For convenience, an additional method accepting a string name of a topic is provided.

unsubscribeOther The method `unsubscribeOther` allows to call `unsubscribe` (see above) on the client whose `PostinaID` is given.

send The method `send` is of key importance in Postina as it is used to send a message directly to a specified client. In contrast to `publish`, messages sent using `send` are delivered only to a single recipient which has to be indicated. Other clients in the network do not receive this messages.

sendReliable Like `send`, the method `sendReliable` is used to send a message directly to a specified client but with the advantage that the delivery is reliable. The use of reliable message delivery implies a higher network load, thus `send` should be used when reliability is not required. Please see Section 4.1.4 for more details about the reliability provided.

addListener & removeListener The methods `addMsgListener` and `removeMsgListener` are used to register or unregister a message listener to Postina. Message listeners have to implement the interface `PostinaMsgListener` (see Section 4.2.3). In case of an incoming message, all message listeners are informed about the new message. There is no difference between the treatment of a direct message (sent using `send`) or a publication (sent using `publish`). Note that registering a listener deactivates the message queue.

`addClientListener` and `removeClientListener` are used to add or remove a client listener, that is a listener that has to be informed when a client is declared dead (see 4.2.4).

getID The method `getID` allows a client to retrieve the current `PostinaID` (see Section 4.2.2) of the current client.

getNextMessage The method `getNextMessage` allows to retrieve the next message in the message queue. If a message listener has registered to Postina, the message queue is deactivated and trying to get a message from it raises a `PostinaNoQueueException`. A description of the exceptions is given in Section 4.2.6. Although clients have to pass a simple serializable when sending a message, messages in the queue are of type `PostinaMessage` (confer Section 4.2.5).

4.2.2 PostinaTopic

The serializable interface `PostinaTopic` encapsulates the topic structure actually used by the underlying implementation. Currently it contains a single method:

getName The method `getName` returns the string name of the topic. This method is mainly for convenience during debugging.

4.2.3 PostinaMsgListener

The interface `PostinaMsgListener` has to be implemented by all clients that want to register as a message listener in order to be informed immediately about an incoming message. There is no separation between incoming direct messages (that is messages sent only to this client) and incoming published messages. The message delivered is a `PostinaMessage` that contains the serializable sent by the originator (see Section 4.2.5).
The interface contains only one method:

receiveMessage The method `receiveMessage` is called to deliver a `PostinaMessage` to the client.

4.2.4 PostinaClientListener

The interface `PostinaClientListener` has to be implemented by all clients that want to register as a client listener. Client listeners are informed when a client is declared dead, that is when a client has left the network.
The interface contains only one method:

clientLeft The method `clientLeft` is called with a `PostinaID` as parameter to inform the listener that the indicated client has left the network.

4.2.5 PostinaMessage

Clients sending a message simply send a serializable. However, some additional information has to be added to the message before it is sent through the network. To make this as easy as possible, the serializable sent by the client is packed into a `PostinaMessage` to which information such as the `PostinaID` of the sender or (in case of a publication) the `PostinaTopic` is added. This information can

then be retrieved by the receiver as the message is delivered as `PostinaMessage` (and not as a simple serializable). The constructors, however, are not accessible to the clients, a `PostinaMessage` can only be created by Postina itself.

The following methods are provided to extract the stored information from a `PostinaMessage`:

getFrom The method `getFrom` returns the `PostinaID` of the sender of the message.

getContent The method `getContent` is the most important function as it is used to retrieve the actual message sent by the originator.

getPriority A `PostinaMessage` can contain an integer representing a priority for message handling at destination. This priority can be extracted at the destination using the method `getPriority`. Priorities are of no significance for the routing in Postina. This feature is provided only so that it can be used by applications using Postina.

getTopic The method `getTopic` gets the topic to which the message was published. If the message is a direct message, the topic has value `null`.

4.2.6 Exceptions

In order to communicate abnormal behaviour after illegal requests to the application, Postina uses exceptions. The following exceptions are used by Postina:

PostinaJoinFailedException A `PostinaJoinFailedException` is thrown if the node cannot join the network of nodes for any reason.

PostinaNoQueueException A `PostinaNoQueueException` is raised when an application tries to get a message from the message queue although the queue has been deactivated. This happens when a listener registers to Postina to be informed about new messages.

PostinaPropertyFileException A `PostinaPropertyFileException` signals that a property file cannot be found or opened.

PostinaPropertyNotFoundException When Postina tries to read a property that is not defined in the property files, a `PostinaPropertyNotFoundException` is raised. Most probably, the property has to be added to the property file. Please see Section 4.4.4 for an explanation of the properties to be defined.

PostinaNotConnectedException When a client which is not connected to the network tries to execute an operation that requires a connection to the network (almost all operations, such as send, publish, subscribe etc), a `PostinaNotConnectedException` is thrown.

PostinaAlreadyConnectedException When a client which is already connected to the network tries to connect a second time, a `PostinaAlreadyConnectedException` is thrown.

4.3 Implementation With Pastry/Scribe

This section explains the internal structure of the default version of Postina, that is Postina using Pastry/Scribe. Although the default version of Postina comes with Pastry/Scribe, any other infrastructure providing both publish/subscribe and direct messaging may be used to implement Postina.

Internally, Postina is split into three different layers. The first layer simply implements the public interface and calls the corresponding methods in the second layer. The second layer is the part providing the additional features and combining the different capabilities of Pastry/Scribe, thus wrapping Pastry/Scribe into the interface of Postina. This second layer accesses the third layer which actually is Pastry/Scribe itself. This way, the implementation details of Pastry/Scribe are perfectly encapsulated and do not matter for the realisation of the other layer.

The UML diagram in Figures 4.2 and 4.3 shows the structure[3] of Postina with Pastry/Scribe. However, the most important classes shall be described shortly in this section, grouped by functionality. All descriptions except those of the special features of Postina are oriented to readers knowing Pastry/Scribe who want to understand how Pastry/Scribe is used in Postina. Please go to Section 4.2 on page 36 if you are looking for a description of the API of Postina.

4.3.1 Main Application

The classes `ScribeNetworkLayer` and `ScribeApplication` are the most important part of the implementation of Postina as they translate the calls of the interface methods of Postina into calls to Pastry/Scribe. The class `ScribeNetworkLayer` implements the main interface `PostinaNetworkLayer` and contains a reference to `ScribeApplication` to which it also registers as a listener. The class `ScribeApplication` implements the interfaces `ScribeMultiClient` and `Application` which are both necessary to run a Scribe application on top of a Pastry node. Additionally, the `ScribeApplication` contains a reference to `ScribeImpl` which is the provided implementation of Scribe.

`ScribeNetworkLayer` prepares the outgoing messages for the transport and then uses the reference to pass all outgoing messages to `ScribeApplication` which will then, in turn, forward them to Scribe. Incoming messages are received by `ScribeApplication` which handles them before delivering them to the `ScribeNetworkLayer` that has registered to `ScribeApplication` as a listener. The `ScribeNetworkLayer` then offers the message to the application using Postina, either by use of the queue or the listener methods. Special request messages that require an action (confer Section 4.1.4) by the node are handled directly by `ScribeApplication` and not forwarded to its listeners. Reliability is also provided by `ScribeApplication`.

[3] For the sake of simplicity Scribe/Pastry is not specified in detail. Please confer [6] for more details.

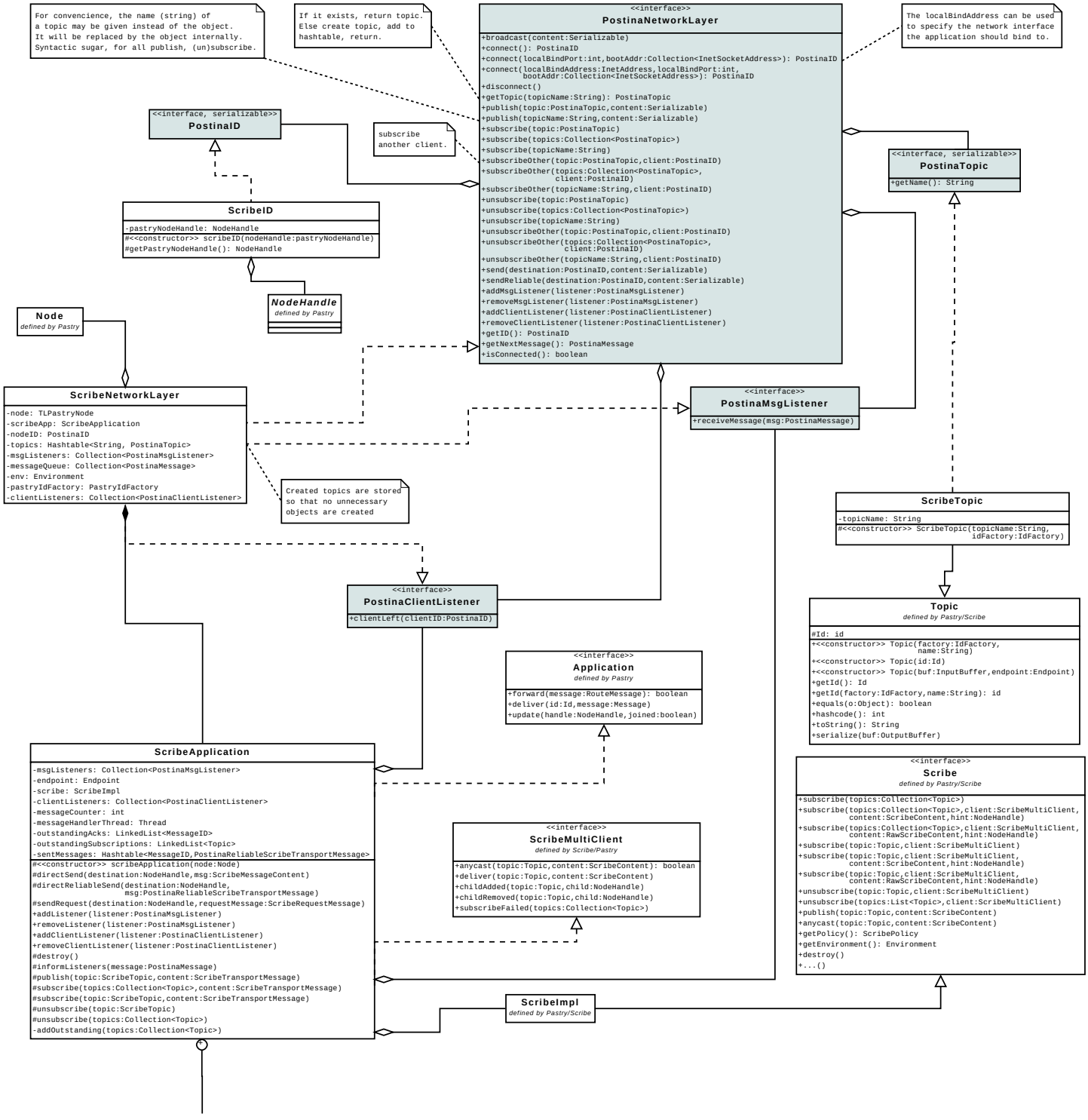

Figure 4.2: The UML diagram representing the structure of Postina, part 1.

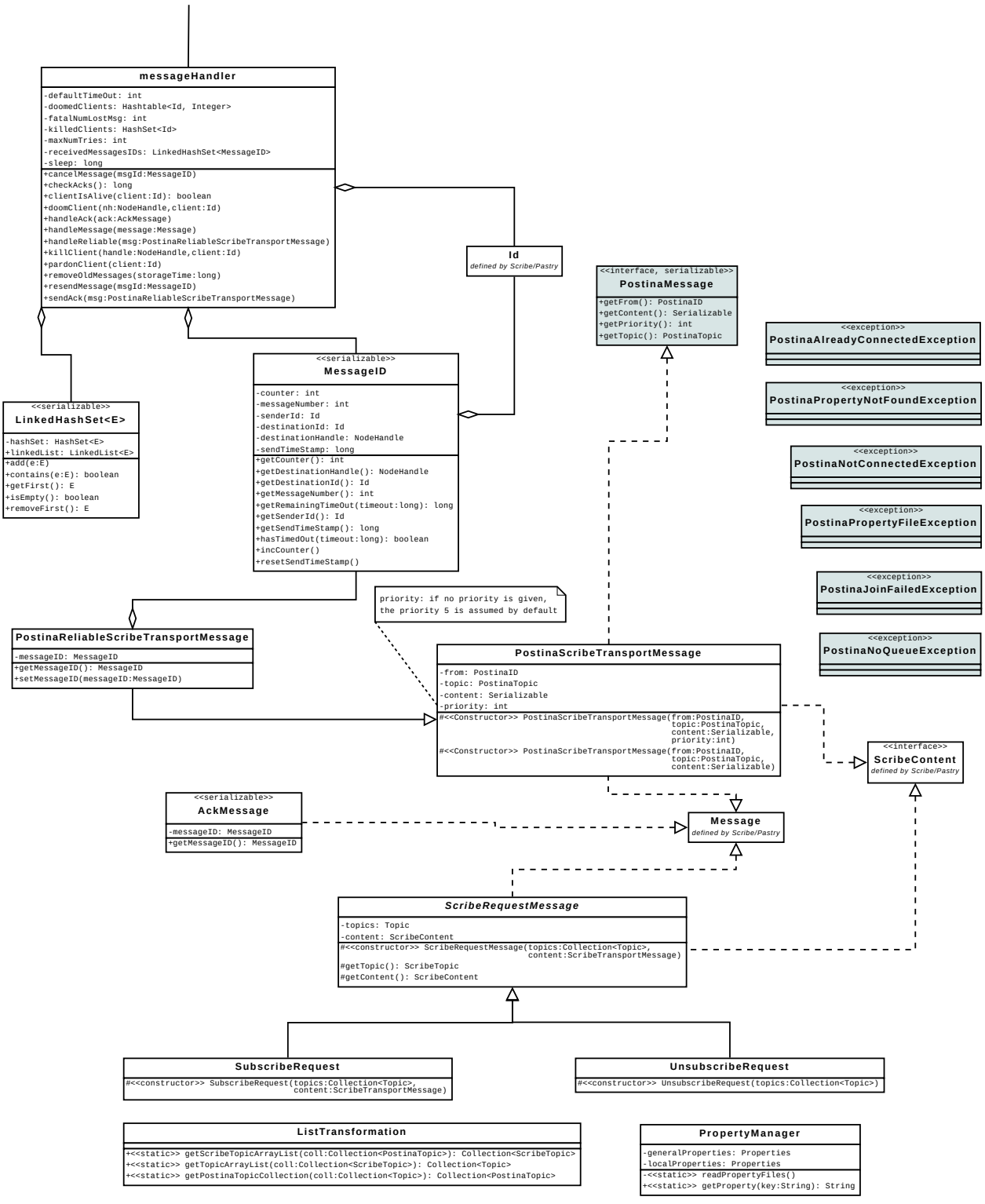

Figure 4.3: The UML diagram representing the structure of Postina, part 2.

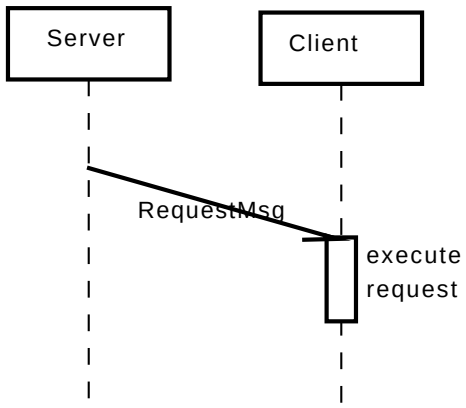

Figure 4.4: Remote calls with Postina.

4.3.2 Messaging

Postina with Pastry/Scribe encapsulates the serializable that the client wants to send into a `PostinaMessage`. This interface is implemented by the class `PostinaScribeTransportMessage` which also implements the Pastry/Scribe interfaces `Message` and `ScribeContent`.

4.3.3 Subscriptions

Postina contains a special mechanism to increase the reliability of subscriptions by keeping trace of subscriptions that have not yet been cancelled and that have not yet succeeded. When Postina is informed by Pastry/Scribe that a subscription has failed, it will re-subscribe to the corresponding topic if no unsubscription has been issued in the meantime.

4.3.4 Identification

The class `ScribeID` implements the interface `PostinaID` and thus provides the unique identifier of a node in the network. `ScribeID` is simply a container of the `NodeHandle` provided by Pastry.

4.3.5 Remote Calls

To implement remote calls, special messages of type `ScribeRequestMessage` and its subtypes `SubscribeRequest` and `UnsubscribeRequest` are used, containing the collection of `Topic` which the receiver should subscribe to or unsubscribe from, respectively. These messages are sent to the client using direct messaging as shown in Figure 4.4. Upon receipt of such a special message, the client takes the appropriate action, that is subscribes to the collection of topics or unsubscribes from the collection of topics.

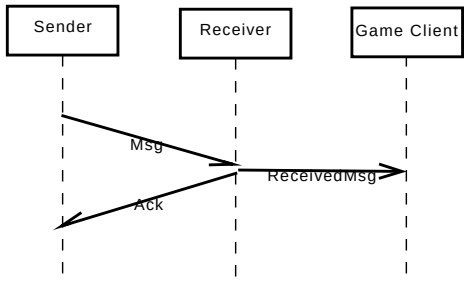

Figure 4.5: Reliability: Behaviour with usual data flow.

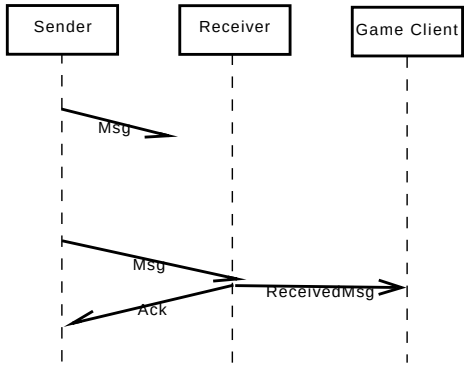

Figure 4.6: Reliability: Behaviour with lost message.

The system for such simulated "remote calls" is built in a way in that it can be extended easily.

4.3.6 Reliability

MMOGs need reliable direct messaging and thus Postina guarantees that a message is delivered (see 4.1.4 for more details). However, Pastry does not provide reliability. Therefore, this feature has to be added on top of Pastry.

To achieve this reliability, a mechanism using acknowledgements is implemented. Sent messages are buffered by the sender. Upon receipt of a message, a client sends an acknowledgement back to the sender (see Figure 4.5) which will then remove the message from its buffer. If the sender does not receive the acknowledgement message within a specified amount of time, it resends the message (see Figure 4.6). Additionally, the receiver keeps track of the messages it has received during a certain period and rejects messages it already has received but still sends back an acknowledgement. This is important when the acknowledgement and not the actual message was lost (see Figure 4.7).

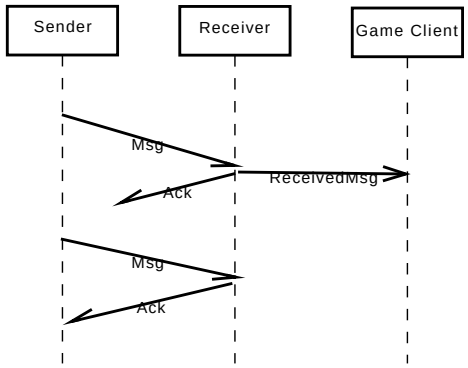

Figure 4.7: Reliability: Behaviour with lost acknowledgement.

4.3.7 Detection of Dead Clients

The detection of a dead client is tricky as a client should never be declared dead wrongfully. Postina uses the reliable direct messaging described above to detect the leave of a client: once the delivery of a message to a client definitely fails, the client is *suspected dead*. When a client has been suspected dead for a configurable number of times, it is *declared dead*. Upon arrival of any message from a client which has been suspected dead, the client is 'pardoned' and not suspected dead anymore.

4.3.8 Broadcasting

The implementation of the broadcasting with Pastry/Scribe is straightforward. After connecting to the network, a node immediately subscribes to a special topic. When a message has to be broadcasted to all nodes, it is simply published to this special topic.

4.4 User Guide

This Section shall give a comprehensive guide for users of Postina. It both addresses the general use and some specific problems that might emerge when using Postina.

4.4.1 How To Use The Provided Version

This section explains how to use Postina with Pastry/Scribe. Postina is designed in a way that its use within another application is very simple. Only a few steps are necessary if you want to use Postina in another project:

1. Add the jar-files for Postina and Pastry to the classpath of your project.

2. Copy the files `postina.properties`, `log4.properties` and `freepastry.params` to a location in the classpath.

3. Adjust the parameters in `postina.properties` (see Section 4.4.4 for a more detailed description of properties).

4. Instantiate the class `ScribeNetworkLayer`.

5. Connect to the network using one of the three `connect`-methods in `ScribeNetworkLayer`. This will create the node and opens the necessary sockets. The operation may take a while. Note that the port defined in the properties has to be available. The method `connect` returns a `PostinaID` which you probably will want to store into a field.

6. If your application has to be informed about a dead client, register it to the `ScribeNetworkLayer` as a `PostinaClientListener`.

7. If you want Postina to inform your application about new messages, register it to the `ScribeNetworkLayer` as a `PostinaMsgListener`. Otherwise you will have to poll the queue yourself.

8. Enjoy Postina and use one of the many methods defined in the interface `PostinaNetworkLayer` by calling them on the `ScribeNetworkLayer`. The available methods are described in Section 4.2.1 on page 37.

In case of troubles, Section 4.4.8 might provide useful hints. Note that Java 1.5 or newer is required to use Postina.

4.4.2 License

Postina has been released under the GNU Lesser General Public License. The license can be found on `http://www.gnu.org/licenses/`.

4.4.3 Multiple Network Interfaces

Many computers, especially servers, have two or more network interfaces. If the wrong network interface is chosen, the Pastry node is not visible to other nodes and the nodes cannot communicate with each other. However, in Pastry all nodes have to be accessible to all other nodes. That is, if the wrong interface is chosen, the Pastry network will not work. For technical reasons it is not possible to have a node listen on more than one interface[4], thus a sensible choice must be made.

By default, the IP address of the network interface is requested from the operating system automatically by Pastry. This means that the choice is random and not necessarily good, possibly leading to a broken system.

[4]By definition, each node in Pastry is bound to an interface. Thus, two nodes would have to be build to listen on two interfaces and hence two 'rings' would be created. MultiRing would have to be used to communicate between these two networks.

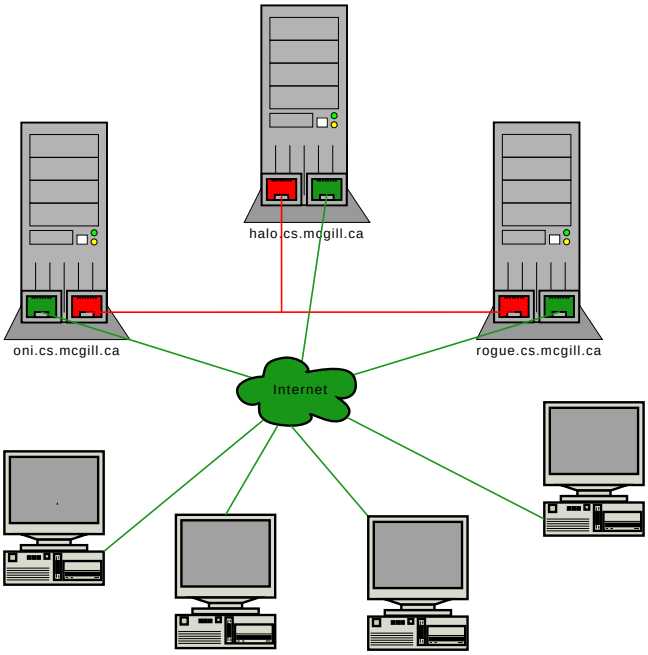

Figure 4.8: The topology of the game research servers.

As an example, we consider the situation at the School of Computer Science at McGill University which is shown in Figure 4.8: There are three servers (`rogue`, `halo` and `oni`) that can be used as Pastry nodes. All three servers have two interfaces, an internal one which is used only for communication between these three servers (represented using red in the Figure) and an external one which is used for communication with all other clients (green). If the internal interface is chosen to bind to by Pastry, the three servers easily build a working network (red lines) but no other node can join this network. We have to use the external interface for the Pastry node (green lines, the three servers can also communicate over the external interface), the internal interface cannot be used.

Solution with Postina

It is not possible to let the application choose the right interface as there is no way to decide automatically which interface is accessible to other nodes. Therefore, this choice has to be made by a human.

The following procedure is used to decide which interface has to be chosen. The tests are given in decreasing priority order:

1. Address of the interface passed to the `connect` method as parameter (see the description in Section 4.2.1 on page 37).

2. Address of the interface defined as property `localBindAddress` in a prop-

erty file (see Section 4.4.4).

3. Address of the interface returned by the operating system.

These procedure is guaranteed to work if there is a network interface present.

4.4.4 Properties

Postina uses the `Properties` of Java and looks out for two different files in the classpath: the default properties stored in a file called `postina.properties` and local properties defined in a file containing the hostname in lowercases in its name. The exact format of the local file name is `postina.<hostname>.properties` where <hostname> is the fully qualified hostname in lowercases of the machine for which these properties should be applied. Example: `postina.se-6.cs.mcgill.ca.properties`

The default properties can be overridden by defining the property in the local property file as a local property has higher priority than a default property.

Format

As the Java properties are used in Postina, the property files have to comply with the format defined in the Java class `Properties` [7]. Each line has to contain a single property which always has the format `propertyName = propertyValue`. Please see the paragraph below for a complete list of possible properties with possible values.

Available Properties

This paragraph gives a complete list of all properties available together with a short description of each property:

bootAddresses The bootstrap nodes to be used when joining the network. *Format:* comma-separated list of `IP:Port`. *Example:* `132.206.3.142:8899,132.206.3.140:8899,132.206.3.141:8899`

localBindAddress The address of the network interface to which Postina should bind. To be used when a machine has multiple interfaces. *Format:* IP address. *Example:* `132.206.51.88`

localBindPort The port that should be used by Postina for the network communication. *Format:* Integer value. *Example:* `8899`

reliability.timeout The number of milliseconds before a message times out. *Format:* Long value. *Example:* `40000`

reliability.maxNumTries The number of times Postina should attempt to resend a message before giving up. *Format:* Integer value. *Example:* `4`

reliability.fatalNumLostMsg The number of definitely lost messages that are allowed before a client is declared dead. *Format:* Integer value. *Example:* `10`

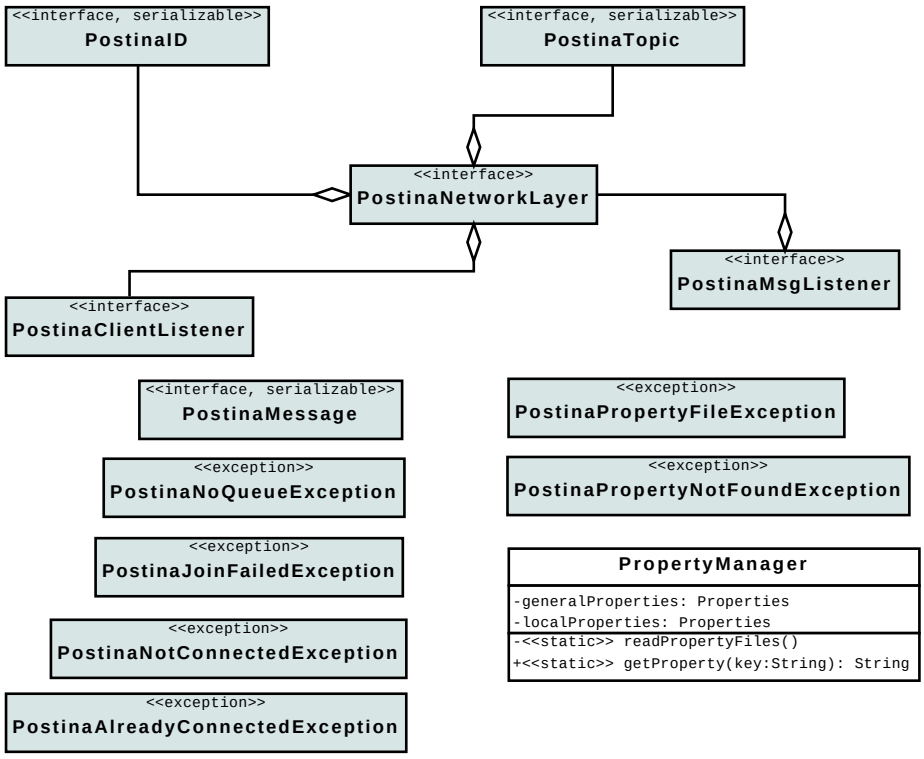

Figure 4.9: The key interfaces of Postina.

4.4.5 Upgrade Pastry/Scribe

To replace the version of FreePastry by a newer one, the jar-file of Pastry simply has to be replaced by a new one.

4.4.6 How To Replace Pastry/Scribe

Although Postina comes with Pastry/Scribe by default, developers are free to choose another underlying system. As the API of Postina is well separated from the effective implementation, switching to another system is very easy. The new system simply has to implement the interfaces defined by Postina as shown in Figure 4.9 on page 50 (see Section 4.2 on page 36 for a description of the API).

In terms of the Java code, all interfaces defined in the package `ca.mcgill.cs.postina` have to be implemented by the new system. Additionally, the package `ca.mcgill.cs.postina.util.properties` has to be kept as it is and the format and naming of the property files has to remain unaffected. In other words, the only code that has to be replaced to use another system is the package `ca.mcgill.cs.postina.scribe`.

4.4.7 Logging

Postina uses `log4j`[3] for logging. By default, logging messages of level `INFO` or higher are printed on the standard output, that is on the terminal. This setting can be changed in the configuration file `log4j.properties` which has to be added to the classpath.

4.4.8 Trouble Shooting

This section gives some important, possibly time-saving, hints to the users of Postina:

- If multiple nodes of Postina are run on the same computer, the same properties are used. Thus, each node tries to bind to the same port. As this is not possible, Postina using Pastry/Scribe automatically binds to the next free port it can find. The used port is indicated in the message printed out when starting up Postina ("Finished creating new node:").
- The local properties override the global properties, that is if a change in the properties does not have any effect, the same property might be defined in a machine-specific file. The use of the properties is explained in Section 4.4.4.
- If you are using Postina in a network using Network Address Translation (NAT) or firewalls please read Section 6.1 on page 59.

4.5 Unit Testing

Postina has been tested using the `JUnit` testing framework. Unit tests are provided to verify the proper operation of Postina. These tests check all interface methods. Special attention was paid to the following criteria:

Message delivery Messages have to be delivered to the appropriate clients. Clients do not unintentionally receive messages, that is publications are delivered only to subscribers and direct messages only to the destination.

Exceptions Exceptions such as the `PostinaNotConnectedException` are only thrown if necessary.

Topic management The creation of topics works.

Queue & Listener Messages can be delivered using a listener or using a queue.

Remote requests Clients can ask other clients to subscribe or unsubscribe.

Connect The connection can be established when the correct parameters are passed. If it fails, the appropriate exception is thrown.

Chapter 5

Application of New Network Middleware in Mammoth

As a proof of concept, the new network middleware shall be integrated into an existing massively multiplayer game. We will use Mammoth to demonstrate the practical application of Postina. The design of Mammoth allows to easily exchange the network engine without having to modify the rest of Mammoth. This procedure is explained in this chapter. First, some preparatory work on Mammoth is documented. Second, the interface of the network layer of Mammoth is explained. Third, the steps to be done to create a new network layer are explained, followed by instructions to actually select another engine. Fifth, the integration of Postina into Mammoth is discussed in some more detail. Last, some experimental results show how scalable the new solution with Postina is.

5.1 Preparatory Work

In order to be able to replace the network layer of Mammoth, some preparatory work was necessary. The existing network layer of Mammoth had first to be cleaned up. All interfaces visible to Mammoth are now in the package `Mammoth.NetworkEngine` and each possible network engine of Mammoth should be packed into a subpackage of `Mammoth.NetworkEngine`.

To actually instantiate a network layer, the factory method pattern is now used. The factory is defined in the class `Mammoth.NetworkEngine.NetworkEngineFactory` and provides three methods:

getNetworkEngineServer() The method `getNetworkEngineServer` without any parameters returns a standard new `NetworkEngineServer`.

getNetworkEngineServer(ServerListener listener) The method

`getNetworkEngineServer` with a `ServerListener` as parameter returns a new `NetworkEngineServer` with a listener registered to it. The listener is informed whenever a new client connects to the server.

getNetworkEngineClient() The method `getNetworkClient` is the only method to get a new `NetworkEngineClient`.

The choice of the type of the server or client returned is taken based on a parameter defined in the properties. Please see Section 5.2.3 for an explanation of how to select a type.

A third major change was necessary to Mammoth to prepare it for the integration of Postina. Up to this thesis, the unique identifier of a Mammoth client was a simple integer. Although this might be sufficient for the existing client/server network layer, alternative network layers such as Postina will probably use more sophisticated identifiers. Thus, this integer identifier has now been refactored into a `NetworkEngineID` that can encapsulate any identifier required.

Finally, the way the server in Mammoth issues subscriptions for other clients has been adapted in two ways to better fit distributed middlewares for the network layer. First, multiple subscriptions for a client that are issued by the server at the same time are packed together into a `Collection` to reduce the number of messages sent. Second, the server keeps track of all subscriptions issued for a specific client. This information is then used to avoid unnecessary subscriptions or unsubscriptions in that a subscription is only issued if a client has not yet been subscribed to the given topic or only unsubscribed if it is indeed subscribed to the topic. This was not the case with the previous network engine.

5.2 User Guide

5.2.1 Interfaces Of The Network Engine

The network engine contains several interfaces that have to be implemented when integrating a new middleware into Mammoth. The interfaces are all in the package `Mammoth.NetworkEngine`. Following, the most important interfaces to be respected when integrating a new middleware into Mammoth are explained shortly:

NetworkEngineClient The interface `NetworkEngineClient` is the most important interface as its methods are those offering the desired functionalities to Mammoth. The client has to be used each time Mammoth is started. Each player is a client.

NetworkEngineServer The `NetworkEngineServer` is responsible for the interest management and the authentication.

NetworkEngineID The `NetworkEngineID` is the new unique identifier of a client in the network.

NetworkMessage A `NetworkMessage` is simply a container for the serializable that has to be sent over the network and for the `NetworkEngineID` of the originator of the message.

5.2.2 How To Integrate Another Middleware

The integration of another middleware into Mammoth is not a complicated task as the network engine has been adapted to ease the replacement of the middleware. To change the middleware, the following steps are necessary:

1. Create a new subpackage of the package `Mammoth.NetworkEngine`.
2. Implement all interfaces of the network engine. See 5.2.1 for a description of the important interfaces to be implemented.
3. Modify the class `Mammoth.NetworkEngine.NetworkEngineFactory`. You will have to add another `else if` to each of the three get-methods defined in this factory class.

Provided that all interfaces are respected, no more changes are necessary within Mammoth. Section 5.2.3 explains how the middleware has to be activated.

5.2.3 How To Select The Middleware To Be Used

Once a network engine has been created for Mammoth, switching from one engine to another is extremely straightforward. All you have to do is to change the property `network.engine` in the property file `mammoth.properties` which is found in the folder `Game`.

Currently the following options are possible for the property `network.engine`:

alf The original network layer of Mammoth using a client-server architecture.

postina The network layer described in this thesis, using Pastry and Scribe.

5.3 Integration of Postina Into Mammoth

Postina was integrated into Mammoth as described in Section 5.2.2 and most of the new code is self-explaining and documented in the JavaDoc. However, some special design decisions shall be explained in more detail in this section.

5.3.1 Client vs. Server

In usual peer-to-peer systems, all nodes are equal and there is no notation of a “server” or a “client”. With regard to Postina, all connected nodes do have the same capabilities. However, Mammoth relies on the fact that there is a server managing the clients. Therefore, a solution respecting both principles had to be found. In the network layer for Postina in Mammoth, a

`PostinaNetworkEngineServer` extends a `PostinaNetworkEngineClient`. A server thus is a normal client but has some additional functionalities and its behaviour on receipt of a message is different from the behaviour of a normal client in that it checks for other types of messages.

5.3.2 Request Message

Request messages are a special type of messages that signal to the receiver that the message has to be treated in a special way and should not be passed to Mammoth as it is. Currently, such request messages are used for two operations:

Blocking Message When a client sends a blocking message it is blocked until it receives a reply message.

Locate Authentication Server At startup, clients do not know where the authentication server is and what its `NetworkEngineID` is. To locate the authentication server, the connecting client broadcasts a special message to all connected nodes. If a server receives such a message, it will reply to the sending client which will then store the `NetworkEngineID` of the server. A message is resent if the client does not receive an answer within a specified timeout.

5.4 Experimental Results

5.4.1 Profiling

To discover potential bottlenecks in Postina and Mammoth, the profiler `JProfiler`[2] has been used. The main focus was on the server used by Mammoth as this application does all interest management for all players and thus has to handle a considerable amount of network traffic. This profiling allowed to analyze the performance of the different parts of the application and to optimize the interaction between different threads, reducing the time the threads are blocked waiting for a lock.

The profiling showed that the reliability added to Postina does not have a significant impact on the performance of the server. At 150 connected clients, the thread handling all incoming messages on the server uses only about 3 percent of the CPU of the server.

5.4.2 Improvements

The profiling allowed to detect and correct several performance flaws and concurrency issues in Mammoth and Postina, including a deadlock situation in Mammoth which arrived frequently with a high number of connected users. Additionally, the profiling showed that Postina with Scribe/Pastry does not have problems with synchronization.

During the tests it became apparent that about 10% of the CPU time was used by Postina to verify if an incoming message had already been delivered

earlier, that is to verify if the unique identifier of an incoming message is in a list of received messages. A deeper analysis showed that this poor performance was due to the fact that a linked list was used to store the identifiers of the received message. With a linked list, the check if an element is in the list is of $O(n)$ where n is the number of elements of the list. As the list of received messages grows quickly despite the fact that old elements are removed periodically, this led to a significant slowdown of the process. For this operation, a hash set with $O(1)$ would be more appropriate. However, the property of the linked list that it supports ordering is of key importance for Postina. Hash sets do not support ordering and thus the linked list cannot simply be replaced by a hash set.

The solution opted for was to define a special data structure called `LinkedHashSet` containing both a hidden linked list and a hidden hash set. New elements added to the list are added to both lists. For all other operations, the more efficient internal list is chosen. This approach was easy to implement and allowed to reduce the CPU time used to handle incoming messages by more than 90%.

5.4.3 Testing with Non-Player Characters

The implementation of Postina with Pastry/Scribe has been tested extensively inside Mammoth using clients with Non-Player Characters (NPC). The behaviour of the NPCs used for this test is rather primitive as the NPCs are simply moving around randomly. However, this behaviour is perfectly appropriate as it causes many messages to be sent over the network and uses both direct messages (to tell the master that the client has moved) and published messages (to inform other players about the change).

The NPC-clients were executed on all available computers in the laboratories of the School of Computer Science at McGill University using a small script that logged in on the machines and then started the NPC clients. FreePastry works best when the clients consecutively connect to the network instead of connecting all at the time. As this behaviour is perfectly realistic for a MMOG, it was simulated using short breaks between the startup of each client.

5.4.4 Capacity

During the tests, the NPC clients were run on about 80 machines in the same network. Up to a total of about 300 randomly distributed clients, the interaction is very smooth and there is no visible difference between the latency with one or 300 clients.

Above 300 clients, Mammoth using Postina is not stable anymore. The server does not react anymore to requests. The profiling analysis showed that this is due to badly synchronized threads in the server application of Mammoth, leading to a situation in which two threads block each other. Thus, it can now be said that the network layer is not anymore the limiting component of Mammoth.

When all clients are in the same interest range, the advantages of publish/subscribe systems become more obvious. The traditional client-server im-

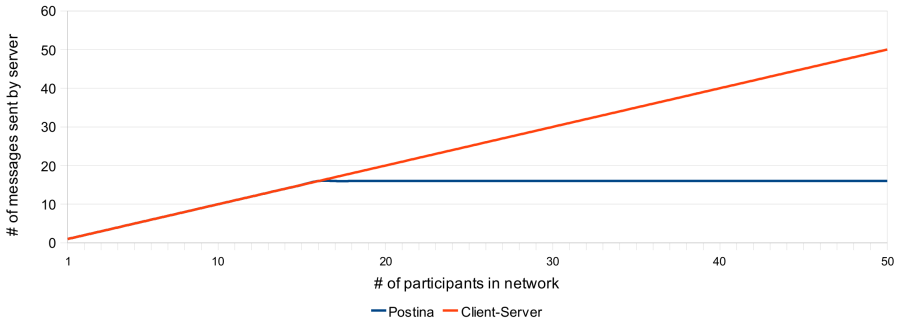

Figure 5.1: Statistical Comparison of Postina ($b = 4$) with Client-Server.

plementation fails as soon as a message has to be sent to about 40 clients whereas a publish/subscribe system such Pastry/Scribe supports fare more clients. Tests showed that about 100 clients within the same interest range do not represent a problem.

Postina with Pastry/Scribe is very efficient when a message is multicast, that is sent to many clients. This is the case when many players are in the same interest range. When the players are widely distributed on the map, the performance gain of a publish/subscribe system over a traditional client-server implementation vanishes. With Pastry/Scribe, a published message has to be sent to the root node and then through the subscription tree to the subscribers, thus publishing a message to very few subscribers introduces an overhead. However, this approach of Pastry is very efficient when there are many subscribers to a topic as shown in Figure 5.1

Number of Messages Sent By Server

Let us assume a network composed by the server and N clients (that is, a total of $N + 1$ nodes). These N clients are all subscribed to the same topic. We now want to compare the number of messages that have to be sent by the server when publishing a message.

In Pastry, the `nodeId`s are created randomly. As explained in Section 2.4.1, each row in the routing table contains $k = 2^b - 1$ routing entries, where b is a configuration parameter with typical value of 4. Each of these routing entries refers to a node. The creation of the lines in the routing tables in Pastry can be assumed to be random due to Pastry's randomization properties.

To compute the number of messages sent by the server node we first have to compute the average number of children nodes per node in the Scribe tree. A node can be part of the Scribe tree without actually being interested in the topic (see Section 2.5.1). Given the method of construction of the multicast tree in Scribe, we can determine in which line of the routing table of a node c its parent node must be referenced. As the lines in the routing tables can be

assumed to be filled randomly, the probability of randomly hitting the routing entry pointing to a node x in the yth line of the routing table of a specific node z is $\frac{k}{N}$. The sample of all possible children nodes of a node is of size N (all other nodes). Thus, the number of times the node x is selected as a parent node is expected to be $N * \frac{k}{N} = k$. Therefore, the average number of children nodes of any node in the Scribe tree is k.

In Scribe, the node with `nodeId` numerically closest to the key of a topic becomes the rendez-vous node for this topic. As the `nodeId`s are created randomly, the selection of the rendez-vous node of a topic is random, too. Thus, the probability that the server node is also the rendez-vous node is $\frac{1}{N+1}$. As shown above, each node has k children nodes on average. Therefore, the server has to send up to k messages on average if it is the rendez-vous node.

With a probability of $1 - \frac{1}{N+1} = \frac{N}{N+1}$, the server is not the rendez-vous node. In this case, it has to send a message to the rendez-vous node to issue a publication to a topic. Additionally, the server node can also be in the tree for this topic, in which case it has to send a message to up to k children on average. Therefore, the server has to send a maximum of $1 + k$ messages on average if it is not the rendez-vous node.

In total, the server node is, on average, expected to send a maximum of

$$\begin{aligned} m &= \frac{1}{N+1} * k + \frac{N}{N+1} * (k+1) \\ &= \frac{k + N * k + N}{N+1} \\ &< \frac{1 + k + N * k + N}{N+1} \\ &= \frac{(1+k) + N * (1+k)}{N+1} \\ &= \frac{(1+k) * (N+1)}{N+1} \\ &= 1 + k \\ &= 1 + 2^b - 1 \\ &= 2^b \end{aligned}$$

messages for a publication with N subscribers when using Pastry/Scribe. Note especially that the maximum number of expected messages sent by the server node is independent of the number of nodes in the network. With the default configuration $b = 4$, the expected number of messages sent by the server node in Pastry/Scribe is smaller than $2^4 = 16$ on average. The experimental evaluation done by the developers of Scribe in [11] show that the measured results are even better when Scribe is tested experimentally.

If the same task is solved with a client-server approach, the server always has to send N messages.

Chapter 6

Limitations of Postina with Pastry/Scribe

Although Pastry is a very strong system and the most appropriate one we could find for MMOGs, it obviously also has some weak points. As the default implementation of Postina uses Pastry/Scribe and relies on its functionalities, the same limitations as for Pastry/Scribe are also true for Postina. In this chapter, the flaws of Postina with Pastry/Scribe will be discussed, namely problems with P2P behind firewalls, reliability issues and finally scalability issues.

6.1 Peer-to-Peer

Peer-to-peer systems are very powerful in terms of scalability. In an ideal peer-to-peer system where all peers are perfectly identical, the load is distributed equally over the entire network of peers. This allows to avoid central servers and thus eliminates a bottleneck. However, the fact that the peers are equal implies that they all have to be accessible from the network. Therefore, firewalls represent a serious obstacle.

During tests of Pastry/Scribe, the fact that it entirely relies on peer-to-peer proved to be problematic. Clients behind a firewall could not join the network and thus built an own partition, that is a 'network' consisting of a single client.

The aim of this section is first to discuss the limitations and second to outline possible solutions.

6.1.1 Problem Description

In this section, first the problems related to P2P existing with the current, standard version of Pastry/Scribe shall be discussed. Second, the necessary configuration to make it work is given.

Limitations

Several tests showed that the current implementation using the standard version of Pastry/Scribe does not work when a client is behind a firewall or in a network using NAT. This means that most users in universities or companies but also many private users cannot connect to the P2P-network and thus would not be able to use a version of Mammoth that uses Pastry/Scribe.

Several tests have been made to find out in which situations the current version works or fails:

- All peers inside the McGill network (software engineering lab computers as well as the servers rogue, halo and oni): this setup works perfectly.
- Client in network at the University of Fribourg trying to connect to node on rogue: fails as rogue cannot open a TCP connection to the client in the network at Fribourg due to firewall restrictions.
- Client on home computer tries to connect to a node on rogue: fails as the firewall prevents rogue from opening a TCP connection to the client.

Requirements for Current Version

Older versions of Pastry/Scribe provided NAT support and configured the firewall automatically using UDnP. However, NAT support is broken in the newer versions. Additionally, UDnP has two serious disadvantages: First, it has to be activated in the firewall. Second, the library used by Pastry does not perform well. Tests made with the home computer failed completely although UDnP was activated on the router. Configuring port forwarding manually failed, too, but was not tested extensively. This negative impression was confirmed by Gabriel Vasile, researcher at the INRIA in France, who faced similar problems.

In order to have Pastry/Scribe work in the current version, the clients are not allowed to be behind a firewall or NAT interface (except the case when all clients are behind the same firewall).

6.1.2 Possible Solutions

Currently there exist five possible solutions to these limitations of Pastry/Scribe:

Ignore The easiest 'solution' would be to ignore the limitations and require all users to manually configure port forwarding, provided that NAT support is re-established.

Super-Peers The problem could be solved using super-peers. Nodes with full access, that is nodes that are not behind firewalls, would be super-peers and build a standard P2P network. Clients behind a firewall/NAT would directly connect to one of the super-peers, following a server-client paradigm. These clients would still be treated as normal members of the network.

Client-server as Fallback System A solution would be to mix the client-server approach with Pastry. By default, a new client would connect to Pastry. If this connection fails, it would fall back to a client-server system and connect to a statically defined client in the network through a direct connection. In the worst case, with all connecting clients being behind a firewall, this solution would thus fall back to the traditional client-server approach. This approach has the advantage that it would be easier to implement than other solutions as no changes to Pastry would be necessary.

Hole Punching Hole punching could be added to the current implementation of FreePastry.

UDP A solution being developed by Luigi De Donà in collaboration with the FreePastry team offers a new network layer for Pastry on the top of UDP which would allow to traverse the firewall/NAT. This network layer, which is supposed to be released soon under LGPL, currently provides the following features (source: e-mail from Luigi De Donà):

1. implements/extends the java nio API
2. includes reliable-sequenced (tcp-like) / unreliable-unsequenced (udp-like) data transfer primitives
3. includes udp hole punching nat traversal, optimized : with inviting caching and superfluous punching pings suppression
4. includes bandwidth control + congestion control tcp-like tcp-friendly
5. uses binary serialization - no java serialization inside, low protocol overhead
6. is content agnostic
7. fragments automatically large packets/buffers
8. is optimized for latency, to transfer small control messages
9. includes a simple centralized mediation server with external address detector
10. includes pseudo NETSTAT command to call via Telnet
11. a server implemented with Jautobahn can accept thousands connections using only one low level DatagramSocket instance.
12. your application can use only one low level DatagramSocket to send and receive messages.

6.1.3 Conclusion

Currently, no ideal solution to this problem with P2P-networks is available. The solution with super-peers sounds promising. However it only works well if many clients are not behind a firewall and function as a super-peer. Additionally,

implementing this solution on top of Pastry might be very challenging and time-consuming. Especially the routing of direct messages could pose problems as clients behind a NAT would not be directly visible to other clients.

The solution proposed by Luigi De Donà would also be interesting and according to Jeff Hoye, a developer of FreePastry, it should be possible to integrate it easily with the existing application. However, it has not yet been released and thus cannot yet be tested.

Jeff Hoye announced that re-establishing NAT- and UDnP-support in FreePastry is one of the most important goals of the developer team. He is currently (February 2008) working on a solution using super-peers. This solution would require about 20% of the clients to be accessible without a firewall. The other clients could then connect to the network using one of these super-peers. Jeff Hoye is planning to add, after completion of the approach with super-peers, a solution using holepunching. By combining these two solutions, all clients should then be able to join the network.

Unfortunately this solution is not yet available at the end of this project. It is, however, very easy to replace the used version of FreePastry with a newer release (see Section 4.4.5 on page 50).

6.2 Reliability

Pastry does not provide any reliability guarantees. Therefore, they had to be implemented on top of FreePastry for Postina as described in Section 4.1.4 on page 36.

6.2.1 Reasons for Loss of Messages

There are several reasons that can lead to the loss of a message in Pastry[1]. First, a published message can be lost when it is waiting in the queue of a node to be forwarded while this node fails. In this case, all messages in the queue of the node are lost and not forwarded anymore. Second, congestion can cause the loss of messages as FreePastry silently drops messages when the queues overflow. This implies that loss of messages is more frequent when many clients are connected to the network and sending messages. Third, the fact that FreePastry drops messages when it doesn't have a current lease for its keyspace from its direct neighbours causes lost messages in case of consistency problems. Fourth, messages may be lost during the process of repairing a subscription tree after detection of a recent node failure.

6.2.2 Limitations of Reliability

The implementation of reliable direct messaging on top of Pastry has the disadvantage of introducing an overhead. Each message has to be acknowledged

[1] See also the message of Jeff Hoye in the FreePastry mailinglist from November 2: `https://mailman.rice.edu/pipermail/freepastry-discussion-l/2007-November/001742.html`.

by the receiver. With no messages lost, the number of messages sent over the network is doubled. Unfortunately, this overhead can hardly be avoided when implementing reliability for Pastry.

Additionally, the implemented reliability is based on end-to-end acknowledgements and cannot provide absolute delivery guarantees as described in Section 4.1.4 on page 36.

6.3 Dead Peers

In peer-to-peer systems, clients may leave the network at any time without announcing their departure. In order to avoid problems, a mechanism allowing to identify dead clients is necessary. Although Pastry internally detects failed nodes, this cannot be used for Postina for two reasons. First, nodes only detect the failure of nodes in their routing table. Second, it may take several minutes to discover such a failure. This is far too long for direct messaging as a client might constantly try to resend all messages to another client for several minutes, which would cause a huge amount of useless network traffic.

The solution for this problem opted for in Postina is to use the reliability mechanism to detect dead clients. The implementation of this approach is described in Section 4.3.7 on page 46.

6.4 Security

Joining the network of nodes in Pastry is very easy. However, this simplicity represents also a disadvantage as there is no access control. Every client wanting to connect can do so. Pastry does not provide any access control. Additionally, all clients in the network can subscribe to all topics and thus also access all information available in the network.

A second security problem with peer-to-peer networks such as Pastry are malicious nodes, that is nodes that seem to be usual nodes but are in fact working against the system's goals. Such nodes might, for example, issue fake publications and thus prevent the system from working well. As described in Section 2.4.1 on page 16, Pastry does not handle such malicious nodes in its default version.

6.5 Scalability

Pastry does have the disadvantage that direct messages have to be routed to the destination through the network of nodes. This procedure is less efficient than sending a message directly to the client without any routing over other nodes. However, it can not be avoided as opening a direct TCP connection between all clients would be far more inefficient and would seriously limit the capacity of the network layer.

Despite the fact that sending a direct message possibly requires many hops on the network, Pastry/Scribe currently does not limit the scalability of Postina within Mammoth as it has been designed for scalability and supports many nodes in the network. Successful tests of Pastry were run with 100000 nodes in a network[17]. The capacity limits of most massively multiplayer games, including Mammoth, are far below this number.

The strength of Pastry/Scribe in terms of scalability clearly lies in the publish/subscribe functionalities. These are especially useful when a message is multicast to many different clients.

Chapter 7

Evaluation

This chapter concludes this thesis by giving a short summary, an outlook suggesting possible improvements and a conclusion.

7.1 Summary

In this thesis we have first studied different available network middlewares. After this evaluation of alternatives, the requirements of MMOGs using Mammoth as an example and the possible solutions with the publish/subscribe systems Padres and Pastry/Scribe were discussed in some detail. This comparison showed that Pastry/Scribe is the most appropriate network middleware for a massively multiplayer game research framework like Mammoth.

After this exact definition of the requirements of MMOGs we designed and introduced Postina, an API for the network layer of MMOGs. Postina combines publish/subscribe functionalities with direct messages and provides some features useful for MMOGs, such as issuing subscriptions for other clients.

In addition to designing Postina we also implemented a version of Postina using Pastry/Scribe and gave a guide for both users and programmers wanting to replace Pastry/Scribe. To be able to test the new framework within a real application, the version of Postina using Pastry/Scribe was then integrated into Mammoth. The following scalability experiments showed that Postina has allowed to considerably improve the capacity of Mammoth and that the network layer is not anymore the limiting factor in Mammoth.

Nevertheless, Postina with Pastry/Scribe still has some limitations which were discussed after the extensive testing. The main problem is the lack of support for clients behind a firewall. Additionally, the reliability provided by Postina with Pastry/Scribe is limited.

7.2 Outlook

Although Postina proved to work well within Mammoth, it can still be improved. Currently, there are three major improvements possible. First, direct messaging is not optimal as messages are routed through the network of nodes which can lead to a large number of hops. A possible project would be to find an optimal mechanism for direct messaging in peer-to-peer networks. Second, the problem that clients behind a firewall are not supported should be solved as this limits the use of Postina with Pastry/Scribe. Third, reliability could be improved so that not only direct messages but also published messages are reliable. However, this improvement would have to be implemented on a lower level, that is directly in the middleware used by Postina, to avoid overloading the network with acknowledgements.

7.3 Conclusion

To conclude, it can be said that Postina with Pastry/Scribe has proved to be an appropriate choice for a massively multiplayer online game. During this project, we were able to significantly improve the performance of Mammoth with regard to the number of simultaneously connected players. After extensive testing it can also be stated that Postina with Pastry/Scribe can be considered as stable. Thus, the task of replacing the network layer of Mammoth with a more powerful one has been accomplished successfully during this master thesis.

The design decisions taken when developing Postina proved to be appropriate during the tests. This allowed to effectively use the new API for the network layer in MMOGs in a real application. Therefore, we have shown that Postina is suited for the use in massively multiplayer online games.

On a personal level, this Master thesis was a success for the author. Thanks to the great support from Alexandre Denault, the author was able to learn and practise many techniques important for practical work as a software engineer. Furthermore, despite administrative obstacles, the experience of doing a part of the studies abroad at another university was very enriching.

Bibliography

[1] The FreePastry tutorial. `http://www.freepastry.org/FreePastry/tutorial/index.html`.

[2] Java profiler - jprofiler. `http://www.ej-technologies.com/products/jprofiler/overview.html`.

[3] log4j. `http://logging.apache.org/log4j/`.

[4] Mammoth. `http://mammoth.cs.mcgill.ca/`.

[5] PADRES: A reliable publish/subscribe middleware. `http://padres.msrg.toronto.edu/`.

[6] Pastry: A substrate for peer-to-peer applications. `http://freepastry.rice.edu/`.

[7] Properties (java 2 platform se 5.0). `http://java.sun.com/j2se/1.5.0/docs/api/java/util/Properties.html`.

[8] Scribe. `http://freepastry.rice.edu/SCRIBE/default.htm`.

[9] J.-S. Boulanger. Interest management for massively multiplayer games. Master's thesis, McGill University, 2006.

[10] M. Castro, P. Druschel, Y. C. Hu, and A. I. T. Rowstron. Topology-aware routing in structured peer-to-peer overlay networks. In *FuDiCo '02: Proceedings of the International Workshop on Future Directions in Distributed Computing*, 2002.

[11] M. Castro, P. Druschel, A.-M. Kermarrec, and A. Rowstron. Scribe: A large-scale and decentralized application-level multicast infrastructure. *IEEE Journal on Selected Areas in Communications (JSAC) (Special Issue on Network Support for Multicast Communications)*, 20(8):100–110, 2002.

[12] M. Coutourier. Flexible pub-sub system for the Mammoth project. Technical report, School of Engineering, McGill University, Montreal, Canada, 2007.

[13] P. T. Eugster, P. A. Felber, R. Guerraoui, and A.-M. Kermarrec. The many faces of publish/subscribe. *ACM Comput. Surv.*, 35(2):114–131, 2003.

[14] E. Fidler, H.-A. Jacobsen, G. Li, and S. Mankovski. The PADRES distributed publish/subscribe system. In *ICFI '05: International Conference on Feature Interactions in Telecommunications and Software Systems*, Leicester, UK, 2005.

[15] G. Mühl. Generic constraints for content-based publish/subscribe systems. In C. Batini, F. Giunchiglia, P. Giorgini, and M. Mecella, editors, *CoopIS '01: Proceedings of the 6th International Conference on Cooperative Information Systems*, pages 211–225, Berlin Heidelberg, 2001. Springer-Verlag.

[16] G. Mühl. *Large-scale content-based publish/subscribe systems*. PhD thesis, University of Darmstadt, 2002.

[17] A. I. T. Rowstron and P. Druschel. Pastry: Scalable, decentralized object location, and routing for large-scale peer-to-peer systems. In *Middleware '01: Proceedings of the IFIP/ACM International Conference on Distributed Systems Platforms Heidelberg*, pages 329–350, London, UK, 2001. Springer-Verlag.

[18] A. I. T. Rowstron, A.-M. Kermarrec, M. Castro, and P. Druschel. Scribe: The design of a large-scale event notification infrastructure. In *NGC '01: Proceedings of the Third International COST264 Workshop on Networked Group Communication*, pages 30–43, London, UK, 2001. Springer-Verlag.

[19] D. Tam, R. Azimi, and H.-A. Jacobsen. Building content-based publish/subscribe systems with distributed hash tables. In *Databases, Information Systems, and Peer-to-Peer Computing*, volume 2944/2004 of *Lecture Notes in Computer Science*, pages 138–152. Springer Verlag, Berlin Heidelberg, 2004.

[20] S. Voulgaris, E. Rivière, A.-M. Kermarrec, and M. van Steen. Sub2Sub: Self-organizing content-based publish subscribe for dynamic large scale collaborative networks. In *The 5th International Workshop on Peer-to-Peer Systems (IPTPS)*, Santa Barbara, CA, USA, 2006.