Supernode-Based Multiring P2P Middleware for the Global Contentifier

Gábor Bernáth, B.Sc.

bernathg@tcd.ie

A dissertation submitted to the University of Dublin, Trinity College,

in partial fulfilment of the requirements for the Degree of

Master of Science in Computer Science

University of Dublin, Trinity College

September 2007

Declaration

I, the undersigned, declare that this work has not previously been submitted as an exercise for a degree at this, or any other University, and that unless otherwise stated, is my own work.

Gábor Bernáth

September 14, 2007

Permission to Lend and/or Copy

I, the undersigned, agree that Trinity College Library may lend or copy this thesis upon request.

Gábor Bernáth

September 14, 2007

Acknowledgments

Many thanks to my supervisor, Dr. Mads Haahr for his advice. The invaluable discussions with the project team members Thom Kubli and Uli Fouquet provided help throughout the project.

A special thanks goes to the NDS class, it has been a great experience. Finally, I want to thank the Ballybough Crew for their support throughout this year.

GÁBOR BERNÁTH

University of Dublin, Trinity College
September 2007

Supernode-Based Multiring P2P Middleware for the Global Contentifier

Gábor Bernáth, M.Sc.
University of Dublin, Trinity College, 2007

Supervisor: Mads Haahr

Abstract

As peer-to-peer applications are becoming ubiquitous for content distribution and communication, modern artists reflect on and incorporate P2P technologies in their work. The novel Global Contentifier installation art project uses P2P technology to drive an interactive audio sculpture interconnecting users and enabling the exchange of personal wishes of political concern, while visualising the geographical and logical network topology and the content flow therein to create an interactive experience. This dissertation strives to realise the technical requirements of the Global Contentifier and is part of a project of larger scale.

A supernode-based P2P middleware for building multiring overlay networks is designed and implemented. The multiring topology consists of the outer and inner rings. The latter interconnects the more powerful supernodes which provide a number of services for the P2P overlay, such as bootstrapping, security mechanisms, as well as geo-location of nodes and content. The design addresses the issues of security, survivability and scalability. An existing ring protocol is adapted to the requirements. The network component of the Global Contentifier application is built on top of the middleware. The designed application realises a serial content

flow within the ring overlay and it communicates with a GUI frontend, which is not part of this dissertation.

Design issues of the middleware and the application are discussed in detail and design choices are presented. A detailed report on the implementation follows, including descriptions of the implementation decisions, the development process and the software architecture.

The results are evaluated experimentally by quantifying performance metrics and overheads. The evaluation discusses the results of several measurements and contrasts the initial goals with the actual achievements. Finally the research is concluded and possible directions of future research are identified.

Contents

List of Tables

List of Figures

Listings

Chapter 1

Introduction

1.1 The Global Contentifier

The Global Contentifier is a project of Thom Kubli, a German media artist. The concepts of the Global Contentifier are summarised here and are described in detail on Kubli's website [1].

The Global Contentifier is a P2P network for distribution of audio content. The audio material is speech or sound expressing personal desires of political concern. The system facilitates the interactive exchange of political views between the users. Users can listen to others and broadcast their own ideas. The Global Contentifier also aims to provide the users with the feeling of *being a part of the whole* by means of geographical and topological visualisation of the overlay network and the content flow therein. A number of network nodes are deployed on "politically interesting" locations in an urban setting, inviting the pedestrians passing by to participate. Home users can download the P2P client and join to the network from the convenience of their own homes[1].

The operational P2P overlay network including the member nodes is an art installation, a constantly evolving audio sculpture that interconnects several public and private spaces and becomes manifest at those locations.

The Contentifier overlay network has a ring-based topology. The topology[2]

[1]The nodes placed around the city will be referred to as "deployed nodes" as opposed to "home nodes".

[2]The geographical topology as well as the logical network topology.

of the deployed nodes initially resembles the shape of the number eight. Nodes are inserted into the network topology based on their *geographical location*. When home nodes join the overlay, the shape of the overlay network evolves, while keeping the eight-shaped topology intact. The new node temporarily replaces the deployed node that is closest in geographical distance. This process is referred to as an *intervention* and accounts for the constantly evolving topology.

The content propagates *serially* through the networked nodes. The resulting circular data flow resembles the repetitions of a mantra. The continuous data flow is ensured even without user interaction. The content is downloaded, stored, played back, forwarded and then deleted automatically. Thus the content is not bound to a specific node or location; instead it moves continuously, traversing the networked nodes without any user control.

1.2 Research Aims

The main aims of this research are:

- to survey the state-of-the-art of ring-building overlays, with a particular emphasis on protocols suitable for P2P systems.
- to design and implement a novel ring-based P2P middleware that provides a platform to realise the Contentifier and other applications.
- to consider security issues and develop countermeasures to minimise threats.
- to produce the networking component of the P2P application based on the middleware.
- to present and evaluate the results of the work outlined above in an informative paper.

The goal beyond the scope of this dissertation is the continued collaboration with the project team to further improve and deploy the Global Contentifier.

1.3 Dissertation Outline

Chapter 1 introduces the concepts of the Global Contentifier project and states the research aims.

Chapter 2 presents essential background information and surveys the state-of-the-art in the field of ring overlays suitable for P2P systems with a focus on multirings and presents applications in the field of geo-based P2P systems.

Chapter 3 enumerates technical requirements, discusses design issues and gives an overview of the main decisions made. The design of the P2P network, the P2P middleware and the client software is presented. The ring protocol is introduced and the behaviour of the supernode architecture is presented in detail. Finally security issues are considered and countermeasures against identified threats are described.

Chapter 4 presents implementation choices including the software and tools used, the protocol stack used for communication, as well as the software architecture. Three main software components are identified, the frontend, the application and the middleware and a detailed report on their implementations is given.

Chapter 5 evaluates the results in light of the research aims and quantifies protocol performance and overheads by means of experimental measurements. A report on implementation status is given.

Chapter 6 presents short and long term future work and summarises the main achievements of the research.

The impatient reader is advised to jump straight to Chapter 3 where all important design issues and decisions are presented and then move on to the conclusion in Chapter 6.

Chapter 2

Background and Related Work

In this chapter, essential background information is presented and the state-of-the-art is surveyed with a particular emphasis on ring topology overlays, P2P systems based on geography, and cryptography in the context of P2P networks.

2.1 P2P Overlay Networks

Peer-to-peer overlay networks construct and maintain a virtual network on top of an existing network, eg. the Internet. The additional logical topology connecting the P2P nodes does not necessarily resemble the underlying physical topology. The overlay network represents a layer between the Internet and the applications running at the nodes. The responsibilities of P2P overlay networks include the management of the overlay itself, routing of messages and providing lookup and search facilities. Overlay management is necessary in a dynamic environment, because the highly autonomous P2P nodes may frequently join and leave the overlay and random link or node failures may occur as well. The overlay network has to be maintained at all times in face of network churn. Nodes act as routers forwarding messages in the overlay network.

There are several benefits of P2P overlays. The overlay abstracts away from the dynamic topology of the underlying network. The overlay network intends to provide basic reliable services to the application layer over an unreliable network. Furthermore, the overlay can exploit the underlying network topology to improve

application-specific performance metrics. Thus, P2P application developers can focus on application development by building on services of the overlay network.

Overlay networks can be used to maintain a range of topologies depending on application-specific requirements. Mesh, tree and ring are common topologies. The latter are discussed in Section 2.2.

2.2 Ring Topology Overlays

A significant amount of research on the topic of ring topology overlays has been conducted. This section discusses general properties of ring networks and presents some of the related work in the field. Sobeih et al. survey, describe and classify several ring-building group communication protocols in [2]. Wang et al. survey and classify multiring protocols for group communications in [3]. These two papers provide an excellent introduction to ring overlays, however, both of them concentrate on military application requirements.

Properties. Ring topology networks share some common properties. Bidirectional rings exhibit inherent tolerance to single node or link failures since there are two disjunct paths between each node. Another beneficial property of rings is that ensuring message ordering is more straightforward than with other topologies. Also, key management schemes are relatively efficient on ring overlay networks.

Rings are scalable in the sense that the node degree and the link stress is constant independently of the number of nodes. On the other hand, network diameter increases with the ring size, which in turn increases delay and jitter, limiting scalability. Furthermore, the throughput of the ring is constrained by the throughput at the node with the lowest bandwidth.

Multirings. The network diameter can be reduced by using multirings, whereby smaller rings are interconnected to form multirings. The rings can be constructed to share either edges or nodes, resulting in a mesh or tree of rings respectively. A few isomorphic varieties of ring meshes are further distinguished in [3]. Both types of multirings can be constructed by recursively appending rings to each other or by creating a shortcut links in a big initial ring. Two metrics of multirings affect the

properties message delay, routing complexity and fault tolerance: the depth of the multiring graph and the number of nodes in the small rings. Multirings increase overall throughput by improving both scalability of bandwidth and latency.

2.2.1 Ring Protocols

FTAR Protocol. In [4] Risson et al. describe Fault Tolerant Active Rings (FTAR), an active topology maintenance protocol for structured overlays. Normal ring membership changes are managed by a variation of the Paxos Commit algorithm. Node join and scheduled leave procedures are implemented as a *transaction* involving the joining (leaving) node and its neighbours. The main contribution of FTAR is that the ring continuity is maintained at all stages of normal membership changes. Paxos Commit achieves consensus in three phases in a non-blocking fashion and guarantees consistency and progress even in the presence of faults. The authors quantify messaging overheads and conclude them to be acceptable. The FTAR protocol is formally specified and proved.

RN Protocol. Shaker et al. present the Ring Network protocol for construction and maintenance of self-stabilising overlay network [5]. The peers running the Ring Network protocol converge from an arbitrary P2P network state to a directed ring topology, ordered according to their node IDs. The proposed protocol is a distributed and asynchronous message-passing protocol and relies only on the existence of a weakly-connected bootstrapping system. All peers execute the simple and robust Ring Network protocol by initiating Closer-Peer Searches periodically and monitoring searches initiated by other peers. The information gathered using the searches is used to update the local successor and neighbour lists.

2.2.2 Multiring Protocols

VRing. In [6] Sobeih et al. present VRing, an application layer multicast protocol, that establishes a multiring overlay in a distributed manner. The VRing is constructed by extending the initial single ring with a spare ring. The spare links create shortcuts between group members. The proposed data delivery mechanism

uses both rings to forward data. The utilisation of the spare ring is shown to reduce the network diameter from $O(n)$ to $O(\sqrt{n})$.

P2P Multi-Ring. Junginger et al. propose a multi-ring topology for high-performance group communication in P2P networks [7]. First an outer ring is formed connecting all nodes. Then, more powerful nodes construct an inner ring among themselves to reduce the network diameter and eliminate their dependency on less powerful nodes. If necessary, multiple inner rings may be established, thereby creating shortcuts to all ring segments. The inner rings are constructed based on local information and no node needs to have a global view of the overlay network. Nodes measure bandwidth to their neighbours and exchange this information in their limited neighbourhood. The multi-ring is designed for dynamic and heterogeneous P2P networks. For example inner ring nodes are replaced if more powerful ones join the overlay. The advantages of inner rings are only useful if the inner ring nodes are well balanced in the outer ring. Thus, inner ring neighbour nodes periodically monitor their distance on the outer ring and swap position with their outer ring neighbours if necessary to improve balancing.

The idea of adjusting the overlay topology to match performance requirements of individual nodes is similar to the more generic concepts described by Haahr et al. [8]. The proposed supernode-based, unstructured P2P overlay network strives to improve performance metrics in a distributed manner. Nodes search for new neighbours with similar characteristics and capabilities to themselves, until the required amount of similar neighbours is reached.

De Bruijn Rings. Wepiwe et al. have constructed HiPeer, a novel concentric multiring overlay topology for highly reliable P2P networks with a bounded number of routing hops [9]. The authors claim that the proposed overlay can maintain the topology in face of a high network churn rate using a bounded number of messages. The overlay constructs a highly connected network topology. The authors tackle the problem from a graph theoretic perspective. HiPeer strives to maximise the number of nodes in a graph with given degree and diameter. The resulting network overlay topology resembles a de Bruijn graph and the number of nodes approximates the theoretically possible Moore bound. The concentric rings

can accommodate an exponentially growing number of nodes, but the size of each ring is limited. When the outermost ring is full, a new ring is added to accommodate joining nodes.

However, the presented routing, resource lookup, node join and node leave procedures are relatively complex. Also, peers need to assist in repairing the overlay before they leave and it is not clear to what extent HiPeer supports node failure. Further research [10] suggests that de Bruijn based topologies (and current constant node degree network topologies in general) fail to meet the conflicting goals of storage load balancing and search efficiency.

Centralised Multirings. There is a wide variety of multirings using some sort of centralised management such as Totem [11] and Hierarchical Self-Healing Rings [12]. However, the lack of distributed management capabilities is not practical in a highly autonomous, dynamic and scalable P2P environment.

2.3 P2P and Art

The Global Contentifier is an installation art project that is based on a P2P network designed for this exact purpose. There has been no published prior work that builds a P2P overlay as an artwork. However, a number of artists use existing P2P networks to distribute their work or to collaborate with others in an effort to create art.

P2P Art [13] is a project by the film director Anders Weberg. The artist made an experimental film entitled Filter for distribution on P2P networks. After completely uploading the film to another user, Weberg deleted the original file and the material used to create it. The film is available as long as users share it. The project emphasizes the "aesthetics of ephemerality". The film is still available on P2P networks at the time of this writing, one year after the it was published. Several other artists produce content for exclusive distribution on P2P networks.

The Electric Sheep distributed computing project [14] generates fractal-like images and animations and display them as a screensaver. It makes use of processing power and human presence at the edge of the Internet. The software uses BitTorrent to distribute the artwork.

It has been reported that a number of musicians use the Skype P2P internet telephony and conferencing service to rehearse and jam online. Others are teaching music through Skype, which allows them to expand their services to a global scale.

2.4 P2P and Geography

A significant amount of research has been done on location-based services and information in computer networks. Interesting concepts and applications have evolved in the field of mobile ad-hoc networks (MANETs), which are related to P2P networks. Unfortunately, it is out of the scope of this dissertation to discuss these in detail.

Most P2P applications that exploit node location in building the overlay network use metrics that rely on the logical structure of the underlying network, such as the round-trip-time and IP address prefix matching. No known P2P overlays build a topology that leverages the geographical location of the nodes, partially because of the difficulty of precisely geo-locating peers.

Geographic Routing. In geographic routing, messages are addressed at a geographic location as opposed to a network address. Messages are routed to the node closest to the destination. Mamei et al. introduce such a geographic routing algorithm for the TOTA middleware [15]. There are several applications in the field of MANETs. For instance geographic hash tables can be built analogously to DHTs, as described by Ratnasamy et al. [16].

Hovering Information. Konstantas et al. introduce the novel concept of hovering information in the context of MANETs [17, 18]. The authors define hovering information as an autonomous entity that is anchored to a specific geographical location, rather than a storage device or physical media. The *active information* is responsible for its own survivability and migrates between mobile devices as they move into and out of the anchor area. The concept behind hovering information is that information is not controlled by the users or devices, but makes an active effort to achieve its goals. The authors briefly mention the possibility that the hovering information could define a *migration plan*, thus moving along its desired path by

making use of the movement of mobile devices and hopping to other devices when necessary. This very interesting research is a work in progress. The introduced concepts are related to that of the Contentifier, where the content traverses along a geographical path.

O'Flaherty proposed the Stirling distributed filesystem [19] for MANETs. In Stirling files are replicated at powerful backbone network nodes near to the file owner's mobile device, thus allowing for low-latency access. The concept of files "following" their owners on the move is somewhat related to hovering information.

Mapping P2P Networks. Pascual et al. developed Minitasking, a tool for visualising the structure of the Gnutella P2P network and search queries propagating between nodes [20]. Minitasking aims to present the logical overlay topology rather than the geographical distribution of nodes.

There are several projects for geographically mapping the Internet backbones and ISP networks. These projects do not aim to locate individual nodes and the techniques used have limited applicability to P2P networks.

Geo-locating IP addresses. Determining the geographical location of computers is important for a number of applications. Several solutions have been proposed. This section briefly introduces the main alternatives. For discussion in detail refer to the survey conducted by Padmanabhan et al. [21].

The simplest solution to geo-locate an IP address is making use of human presence. Many applications ask the users where they are located. This solution has several problems, as it depends on the cooperation and honesty of the user. The other major client-side solution is the application of GPS receivers or other equipment capable of locating itself, but today few users have such devices.

Numerous databases and webservices exist for mapping IP addresses to geographical locations. A problem with existing services is that often times they are either expensive or inaccurate, or both. However, for many applications they pose the most viable alternative. IP-to-geo services are often used by geo marketing and fraud detection systems. Many P2P file sharing clients use these methods to infer the country or organisation where the peers are located in order to avoid logging the transmission of copyright infringing content by law enforcement.

Other methods are based on running traceroute, or using triangulation to infer the location by measuring propagation delays from a number of servers deployed worldwide at known locations. There are solutions that combine multiple methods to increase accuracy.

2.5 Cryptography

This section gives an overview of cryptography in the context of P2P networks.

2.5.1 Encryption and Signatures

Secret key or public key cryptography can be used to address the security threats of eavesdropping, insertion and modification of messages.

Symmetric Cryptography. Secret key (symmetric) cryptography shifts the security issue to the one of key distribution. One possibility for using secret key cryptography in group communication is to distribute a single secret key among the group members. Apart from the contradiction of distributing a secret, the need of rekeying the whole group arises on every membership change. Leaving nodes should not be able to continue decrypt the communication and joining nodes should not be able to decrypt messages recorded beforehand. Such a requirement is most infeasible in P2P overlays with a high network churn.

An alternative is establishing shared keys for each link independently. This alternative is manageable in a ring topology overlay with constant node degree. However, it is hard to agree on a shared secret without mutual authentication. For example the Diffie-Hellman key exchange protocol is vulnerable to man-in-the-middle attack if the two parties cannot authenticate each other. The parties need to have a shared secret in advance or a public key infrastructure (PKI) or similar mechanism in place. Nodes to not have a shared secret in a P2P system; the possibility of PKI deployment is discussed below.

Asymmetric Cryptography. Public key (asymmetric) cryptography can be used to encrypt and sign messages. The main difficulty in deploying public key cryptography is to find a way to distribute public keys in an authentic fashion. This

can be achieved by delegating trust to a third party. The trusted third party is most commonly a certificate authority (CA), which issues a certificate that binds a public key to an user identity[1]. A classic public key infrastructure automates the process described above in a centralised fashion. The "web of trust" used by PGP is different way of creating similar certificates in a decentralised manner.

Centralised PKI. Most PKI systems rely on a hierarchy of certificate authorities. The certificate of an end user is issued (signed) by a CA. The identity of CAs is also described in a certificate issued by a higher-level CA. Thus, the identity of an end user is described through a certificate chain. The root certificate used to establish the identities of the high-level CAs is distributed to the end users by out-of-band means, usually together with the software performing validation of certificates. The identity of an user can be proven by following and verifying the certificate chain up to the root certificate. Centralised PKIs make use of a directory scheme, such as LDAP, to store certificates in a hierarchy.

The traditional X.509-style PKI has a number of shortcomings, including complexity, rigid and impractical directories, naming problems and revocation issues [22]. While many of these shortcomings can be addressed by omitting the directory and choosing another means of distributing and managing certificates, revocation issues still remain. Certificate revocation lists (CRLs) have proved inefficient; online revocation solutions create a bottleneck and single point of failure at the server. Issuing short-lived certificates shifts the problem to revalidation and requires synchronised clocks, which contradict the self-governing nature of the Internet. As an effect, several applications design around the problem of revocation, such as SET and SSH.

Distributed PKI. In [23] Datta et al. present a quorum based distributed PKI built on top of a P-Grid, a structured P2P system with a tree-based key-space. Their proposed system replicates public keys at multiple nodes. A node looking for a key retrieves it from a set of replicas. Given a set of answers the key originally stored in the overlay is deduced using statistical methods. Thus the information is available even in the presence of failures and malicious nodes. The authors

[1]In the context of the Contentifier the identity is described by the node ID, which includes the IP address, port number and geo-location.

advocate the quorum based approach over PGP like web of trust models because of the possibilities of exploiting collective knowledge and quantifying probabilistic guarantees. The proposed system could be adapted to a DHT-based overlay.

Wöfl et al. propose P2P-PKI [24, 25], a distributed PKI based on Chord. It relies on trust-based metrics to provide authorisation. P2P-PKI distributes trust management like PGP, but goes further and distributes the directory as well. P2P-PKI is a work in progress and is missing certificate revocation and expiry features.

2.5.2 Pricing

Pricing is a mechanism that allows a service provider to force clients to prove they are serious about a request. The client contributes a small price for each request and the service provider verifies the contribution before processing the request. Pricing can be used to mitigate or at least slow down large scale attacks such as spamming, distributed denial of service and sybil attacks. The price can be monetary or computational.

Micropayment. Micropayment systems facilitate the transfer of very small amounts of money, thus large scale attacks get expensive. However, micropayment systems involve real money, require a considerable administrative effort and are impractical in open P2P systems, where users are not willing to open their wallets.

The alternative "payment" method involves trading computational power by solving cryptographic puzzles, which is analogous to minting electronic coins or stamps. The idea behind crypto puzzles is the same as in micropayments: it is expensive to mint large amounts of electronic coins, but it takes a very short time to mint a single one.

Hashcash. Back proposed the hashcash system as a denial of service countermeasure [26, 27]. Hashcash requires the client to find a string that has a corresponding SHA-1 hash with the first x bits set to zero. The amount of computational work required can be parametrised by choosing an appropriate value for x. The generated string is easy to verify by the receiver. Attackers could compute many stamps in advance of mounting an attack. Another issue is double-spending. There are several variations on the hashcash cost-function including interactive

ones, which solve the problems of computing stamps beforehand and double-spending. Others have raised concerns about the effectiveness of crypto puzzles, because it is hard to parametrise them in a way that is acceptable to the slowest legitimate node but sufficient to slow down attackers with access to a high amount of computational power [28]. The attacker has to consider a cost-benefit trade-off depending on the value of the service attacked. In an open P2P system where user selfishness is rather a motivation for attacks on the network, cryptographic puzzles are an effective way to deter an attacker.

Chapter 3

Design

This chapter enumerates the technical requirements of the Global Contentifier, discusses design issues in detail and presents design decisions. The design of the overlay network, the supernode services, the software architecture and security measures are addressed.

3.1 Design Choices

3.1.1 Requirements

The middleware requirements of the Contentifier differ from that of other P2P applications in many aspects. This section enumerates requirements that affect the overall design. The implied design choices are discussed in the following sections.

The main requirements of the Contentifier are:

- build and maintain the ring-based topology
- serial content flow in the ring
- allow home users to participate in the network
- enforcement of overlay policies affecting node join and content insertion
- geo-location of nodes and content and aggregation of a global view of the overlay network topology and the content flow

3.1.2 P2P Overlay

Serial Content Flow. Most P2P applications are sensitive to message delay, therefore, most ring protocols aim to reduce latency at the cost of networking and storage overheads. Multirings have been introduced to improve scalability of message delays as the number of nodes in the network grows. Latency is mitigated by enhancing connectivity to distant ring segments, that in turn decreases the network diameter.

The Contentifier application in contrast intends to distribute content in a sequential fashion with an inherent propagation delay. This relaxation of requirements greatly reduces complexity.

Global View. A requirement of the Contentifier is to present a global view of the P2P overlay network topology to the user. Distributed group membership management protocols do not aggregate a global view of the overlay, P2P nodes are only aware of their limited local neighbourhood. The Contentifier needs to preserve scalability of the network without exposing a single point of failure. Thus, the ring management protocol has to be kept distributed and the aggregation of a global network view has to be realised at a higher level. Implementing such functionality at a higher level keeps the ring protocol scalable and shifts the bottleneck to the higher level.

Home Users. A requirement of the Contentifier is to allow home users to download the client and participate in the P2P system. In this context two conditions have to be met: platform independence and network connectivity.

The users running different operating systems on a variety of devices should be able to run the client without an effort. This implies the need for cross-platform solutions to keep development effort low as well. The programming language and libraries used have to support different platforms.

The other issue to be addressed is network connectivity. Nowadays, home users are likely to be located behind home routers and middleboxes of other kind performing Network Address Translation. NAT violates the end-to-end argument [29] and causes difficulties for P2P communication. In practice, many P2P applications use the controversial UPnP protocol to work around middleboxes

by enabling *port forwarding*. Another widely used NAT traversal technique is *hole punching* which in general requires third party coordination services in the initial phase of setting up a connection. Hole punching is possible with both UDP and TCP network transports, while devices supporting UDP hole punching are more numerous [30].

The Contentifier protocols are based on TCP communication to make use of its advantages in reliability. Implementing the required TCP functionality over UDP is an error prone and tedious effort[1].

The requirement of the intervention, whereby home nodes *replace* deployed nodes, imposes a constant number of nodes. This requirement is not addressed by the middleware, which should be able to scale to a potentially large number of nodes. Instead, the restriction is implemented as an application layer policy.

Supernode Architecture. Several requirements of the Contentifier need services that are centralised to a certain extent. These requirements include maintaining a global view of the network, aggregating content flow tracking information, geo-locating peers and content, enforcement of policies and key management. Providing these services from a single central server is not feasible because of the limited scalability and failure tolerance of such an architecture[2]. Other practical concerns such as the lack of funding for providing high bandwidth and processing power in a single location are against the deployment of a completely centralised architecture.

However, the Contentifier network will have a set of nodes deployed in booths around the city. These nodes are arranged in a relatively static topology and are under control of the network operators. Thus, a supernode based architecture is a viable alternative to support the required services in a decentralised manner.

The design of the supernode architecture is elaborated in detail in Section 3.1.5.

[1]Even tough some work has already been done in the Twisted community: see pseudo-TCP or PTCP [31].

[2]At the time of this writing the prominent supernode-based P2P telephony service Skype has experienced a two-day outage due to a massive amount of simultaneous login requests at a centralised server.

3.1.3 Software Architecture

Application Architecture. The Contentifier software has two main components: the frontend and the network component. The frontend is an user interface that allows the user to produce and consume content and visualises the global view. The network component is responsible for managing the P2P infrastructure. It is composed of the middleware and the application. The middleware layer provides maintenance of the overlay network, while the application layer relies on the overlay network to distribute content and aggregate the global view information. Figure 3.1 illustrates the components. The scope of this dissertation is the network component, shown in grey.

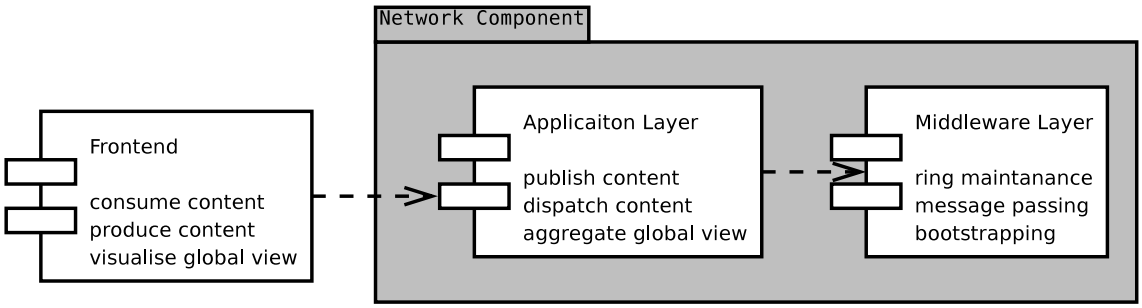

Figure 3.1: Application Components

Protocol Overview. The communication protocols for the Contentifier use a layered protocol model. The protocol layers provide services to the layers above and depend on services of the underlying layers. The lower protocols provide message-based communication and related abstractions. The ring protocol layer provides membership management, ring maintenance and failure handling. The protocol layers above the ring protocol are collectively referred to as application layer protocols. These realise content flow control and global view aggregation. The supernode functionality crosscuts several protocol layers[3]. The protocol stack is shown in Figure 3.2. The design issues of the different layers are addressed in the following sections.

[3]For example the supernode protocol aids the ring protocol layer with providing bootstrapping functionality and also drives the aggregation of the global view.

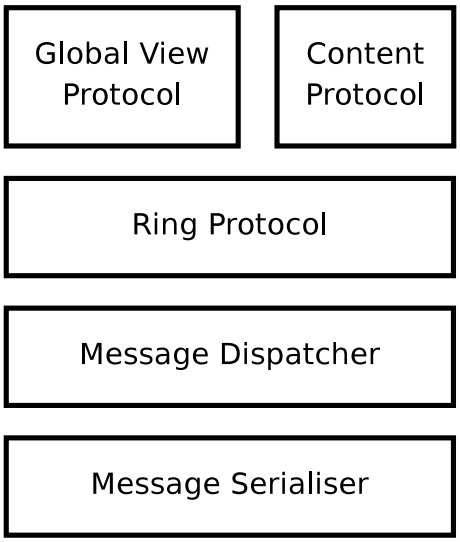

Figure 3.2: Protocol stack

3.1.4 Ring Protocol

The ring protocol for the Contentifier overlay network is in many ways similar to existing ring protocols, in particular VRing [6], which it is based on. Nodes are arranged in a ring. Each node keeps track of its neighbours, that is its immediate *successor* and *predecessor*[4]. However, nodes may fail at any time, breaking the ring. It is desirable to keep track of a number of successors[5] and predecessors. Hence several neighbouring nodes may fail simultaneously without breaking the ring ultimately. The designed ring protocol can cope with a configurable amount of failure.

Ring Maintenance. The nodes in the ring have to maintain the ring continuously. For this reason each of the member nodes runs the *hello protocol* with its successor. The hello protocol is a heartbeat mechanism that allows to detect node failures. The hello protocol compromises sending `Hello` messages periodically to the successor

[4]Some authors refer to the successor as the "downstream node" and call the predecessor the "upstream node".

[5]The term "successor list" will be used in this paper to refer to the list of known successors. The "neighbour set" refers to all known successors and predecessors, that is the successor list and the predecessor list.

and monitoring the arrival of `HelloReply` messages or the lack thereof. When a node detects the failure of its immediate successor, it attempts to repair the ring. On receiving a `Hello` message the node replies with a `HelloReply` message if the `Hello` originates from its current predecessor.

The hello protocol also has another purpose apart from failure detection. The nodes include their successor list with each `HelloReply` message. Using this mechanism the nodes get to know the desired number of successors. The nodes build up knowledge about their limited neighbourhood in a completely distributed manner.

Failure Handling. When a node detects the failure of its successor using the hello protocol as described above, it initiates a recovery process to repair the ring. It sends a `LinkRepair` message to the successor of its successor (2 hops away) and waits for a `LinkRepairReply` message. If no `LinkRepairReply` arrives, the node assumes that its two immediate successors have failed and contacts its next successor (3 hops away). This process is repeated until a `LinkRepairReply` message is received or there are no more known successors left.

Consequentially the node either succeeds in repairing the ring or comes to the conclusion that the ring is broken beyond repair, leaves the ring and restarts the bootstrapping phase.

Join Procedure. A node n intending to join the overlay first contacts a supernode and retrieves the node ID and address of one or more peers. This process is called *bootstrapping* and happens before the *join procedure*, which is described here.

The node n sends a `Join` message to a peer p. The `Join` message contains information about n's current neighbours if it has any. The join procedure will result in either:

- a ring of two nodes (if the peer p was isolated), or
- a larger ring (if p was already in a ring) comprising of the original ring and the new node n, in such a way that p is the successor of n and n is the successor of p's predecessor in the original ring, or
- failure (if p decides not to participate). In this case no changes take effect.

On receiving a `Join` message the peer p decides[6] if it wants to participate in the join operation. Any node can participate in a single join procedure at a time, otherwise network partitioning may occur. Based on the decision, p either ignores the `Join` message or it sends a `JoinReply` message to the requesting node n. The peer p includes its neighbour set in the `JoinReply` message. If p was already in a ring, it also notifies its old predecessor of the topology change by sending a `JoinTo` message that indicates the node ID and address of n.

The join procedure involves two or three nodes, as described above. Along the procedure the affected nodes update their neighbourhood sets to point to the new neighbours.

Differences to VRing. There are a number of differences between the ring protocol proposed above and VRing due to the different requirements. VRing is an application layer multicast system, while the Contentifier is a content distribution system.

VRing relies on a centralised *Rendezvous Point* (RP) as a bootstrapping mechanism. Each node has to contact the RP before joining the ring. The proposed ring protocol uses a decentralised bootstrapping service that is provided by the supernodes.

After establishing the overlay ring, VRing constructs an additional *spare ring* comprising all nodes to reduce the latency of message propagation. The VRing protocol relies on the RP to determine when to start the construction of the spare ring and needs to know the total number of ring members to do so. The Contentifier does not make an effort to reduce message propagation delay, because the sequential propagation of content is a core concept. Instead, the supernodes build an inner ring to reduce the latency of *control messages* among themselves. Regular nodes do not participate in the inner ring. Both the inner and outer rings are maintained using the proposed protocol.

The third difference is that VRing introduces the concept of *leader nodes*. In the overlay construction phase, each ring has a leader. Two leaders negotiate to join the rings together and at the end of this process one leader retires. At the end

[6]The decision involves the verification of the node ID of node n, the neighbour set of p and its current state.

of the overlay construction phase a single leader remains in the ring that initiates and controls the spare ring construction. The VRing protocol requires a distributed monitoring and additional election algorithm to handle the failure of ring leaders. In the Contentifier the work of the leaders is delegated to the supernodes, without exposing regular nodes to increased responsibility.

3.1.5 Supernode architecture

The Contentifier has requirements for a number of services. Care was taken not to centralise any services for obvious reasons. However, completely distributing these services would increase design complexity and expose all nodes to increased responsibility and security threats. The supernode architecture was chosen to cope with the mentioned requirements by decentralising services, rather than completely distributing them. In this section the supernode architecture and the services provided by supernodes are described in detail.

The following services are provided by supernodes:

- bootstrapping
- node ID generation
- geo-location of nodes
- management and enforcement of policies concerning node join and content insertion
- aggregation and distribution of the global view and content flow information

Inner and Outer Rings. Both supernodes and regular nodes are connected in a single ring, the outer ring. The supernodes also participate in an inner ring. The inner ring provides low-latency communication for exchanging information about the global network view and the content flow, without using any resources of the regular nodes. Content traverses the outer ring and control messages along the inner ring.

Figure 3.3 illustrates the two rings. The supernodes (nodes 1, 4 and 6) are connected with an additional inner ring.

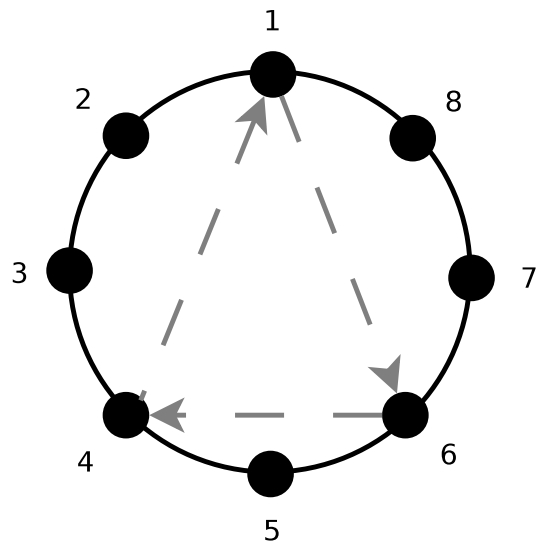

Figure 3.3: Inner and outer rings

Bootstrapping and Node ID Generation. Bootstrapping[7] is the process of aiding new nodes to discover nodes that are already members of the overlay. Bootstrapping is necessary in P2P networks because of the lack of fixed nodes that can be assumed to be a member of the overlay and because of the dynamic changes in the IP addresses of participating nodes.

Node ID generation is the assignment of a globally unique identifier (GUID) to a node. The node ID can be regarded as the address of the node within the overlay network. The node ID also determines the location of the node in the logical network topology in many P2P networks[8].

New nodes intending to join the overlay require a node ID and bootstrapping information. The new node contacts a supernode to satisfy these needs. It is expected to know the IP address of at least one supernode or obtain it by means of DNS or another out-of-band method. The joining node selects a random supernode and initiates a *registration process*. During the registration process the new node is geo-located by a supernode, a node ID is generated and signed by the supernode. The new node receives its node ID along with a list of node IDs of geographically

[7]Some authors use the term "rendezvous" for bootstrapping.
[8]In particular DHT-based P2P networks.

nearby peers.

Content Insertion. The process of inserting new content into the overlay also involves supernodes. Supernodes can control the amount and rate of content insertion on the overlay. A node trying to broadcast new content has to ask for permission from a supernode. This involves calculating a hash[9] of the content itself, providing some additional metadata, as well as solving a crypto puzzle based on the content hash.

The supernode considers current network load and overlay policies and decides about giving or denying permission for the new content. If the operation is permitted, the metadata is timestamped, signed and sent back to the requesting node. Then the content can be distributed without supernode interaction, as the metadata is self-certifying.

The structure of the metadata is shown in Table 3.1.

Attribute	Description
id	Content identifier, the hash of the data
name	Original file name, optional
size	Content size in bytes
pieces	Number of content parts
piece_size	The size in bytes of each piece
length	Content playback duration
codec	The codec used to encode the content
created	Timestamp indicating creation time
signature	Digital signature of the metadata

Table 3.1: Attributes of the `Content` class representing the metadata

Global View. The aggregation and distribution of the global view of the overlay topology and the content flow therein is the responsibility of the supernodes. In providing this service the supernodes depend on regular nodes reporting network events.

Each node has a supernode assigned to it. The assigned supernode is the next supernode in the outer ring that is downstream from the node. That is, the

[9]The hash is calculated using a collision-resistant cryptographic hash function.

node's assigned supernode is the first supernode among the successors of the node in question. Each supernode is responsible for aggregating information about the topology and content flow in the ring segment containing the nodes assigned to it. In other words, this is the ring segment including its predecessors up to the next upstream supernode.

Each node reports two types of events to its assigned supernode: topology-related and content-related events. A topology-related event is a change of the successor. This happens when a new node joins the overlay or when a member of the overlay fails. Content-related events are the beginning and ending of content downloads. The supernodes track the current geo-location of all content in the overlay network.

Supernodes use the inner ring to exchange information about their ring segments. In this manner all supernodes have up-to-date information about all ring segments. The global network view and content flow information is broadcast along the outer ring segments to each node, to be presented on the screen to the user. Transmitting the whole view to regular nodes each time the topology or content flow changes would create a large overhead. Instead the global view is transmitted initially when the node joins the ring and updates are distributed that allow the regular nodes to refresh their view of the topology.

Assumptions. This section lists the assumptions about the supernodes.

- Supernodes have sufficient knowledge about each others network addresses to create a weakly connected network.
- Supernodes do not exhibit malicious behaviour.
- Supernodes have a node ID and public key which is signed by the network operators[10].
- Supernodes have synchronised clocks[11].

[10]The public key of the operators is distributed along with the client software.
[11]A practical way of synchronising clocks is running an NTP service.

3.2 Security Considerations

Before considering appropriate security measures In this section, first the different node roles and communication characteristics in the Contentifier overlay network are introduced. Following that, threats are evaluated and possible solutions are discussed.

There are two types of nodes in the Contentifier network: supernodes and regular nodes. All nodes run the ring management protocol and the Contentifier application. Supernodes run addtional services the overlay network depends upon.

From a security perspective, it is important to consider the types of communication and the kind of data exchanged. Communication between the nodes can be classified as follows:

node to node Nodes announce, request and forward content and associated metadata to each other. Also, information broadcasted by supernodes is forwared between regular nodes.

node to supernode Nodes request bootstrap information when joining the network. Supernodes geo-locate nodes, assist in ID assignment and communicate this information to nodes along with policy configuration that has to be obeyed by the nodes. Furthermore, supernodes distribute information about the global network view and content flow. Nodes report content flow information to supernodes.

supernode to supernode Superndoes exchange bootstrapping information to be sent to joining nodes, as well as information about the global network view and content flow. The transmitted data includes node IDs, IP addresses, geo-location of nodes and content, content metadata and other statistical information.

CIA Triad. Confidentiality, integrity and availability are widely accepted as the three key components of information security and are also referred to as the CIA triad. Confidentiality is achieved by providing data privacy, that is preventing unauthorised parites from reading the data. Integrity is ensured by preventing

unauthorised modification data. Availability means that authorized users can reliably access the data at all times.

There are several security threats that compromise one or more of the security components mentioned above: eavesdropping, insertion, modification, replay and deletion of messages. These threats have to be addressed in order to achieve secure communication in the P2P network.

3.2.1 P2P Security

In an open P2P system nodes cannot be generally trusted. In such an environment failures are common and nodes might exhibit malicious behaviour. The P2P network has to operate in the presence of faulty and malicious nodes.

Extensive research has been done on the topic of P2P security. Engle et al. present vulnerabilities in current P2P systems and evaluate possible solutions [32]. The authors identify lower layer attacks and P2P layer attacks. Denial of service (DoS) and man-in-the-middle (MitM) are classified as lower layer attacks, while user selfishness, sybil and eclipse are P2P layer attacks. Their results are summarised here. DoS can be limited using pricing. MitM can be prevented by distributing services or using public key cryptography. User selfishness is usually solved using a tit for tat strategy. The sybil and eclipse attacks can be effectively addressed by limiting the node join rate, expiring node IDs, and preventing the attacker from choosing its own node ID[12].

Security Measures. These results are applied in the design of security measures.

Small scale attacks are be prevented from corrupting the overlay as a whole by increasing the availability. Availability is achieved through scale and distribution. Failure of single nodes must not compromise the P2P overlay. The desired degree of availability is achieved by designing a ring protocol that tolerates a configurable amount of failure (trading off bandwidth and storage).

Large scale attacks are mitigated by using pricing and secure node IDs. Node IDs are signed by a supernode and given a short expiration period. Hashcash is used in the Contentifier to limit the rate of node join and content insertion attempts.

[12]Contentifier nodes are inserted into the topology according to their geo-location as opposed to the node ID. That said, choosing the node ID translates to choosing the geo-location.

The client has to provide proof-of-work before joining the overlay network and to insert new content into the overlay.

User selfishness can be addressed by enforcing overlay policies regarding bandwidth requirements and content insertion. The constrained serial data flow mitigates the benefits of user selfishness.

3.2.2 Cryptography

This section discusses cryptography in the context of the Contentifier. Related work has been discussed in Seciton 2.5.

Encrypting and signing of messages prevents outsiders from eavesdropping, modification and insertion of messages. Replay and deletion of messages have to be addressed separately. However, all information in the overlay is public inside the overlay. An attacker could join the overlay at any time to compromise confidentiality through eavesdropping or message insertion. In an open P2P network anyone should be able to join the overlay and listen to the broadcast content, thus there is no point in encrypting messages. Integrity has to be ensured of course, but confidentiality is not a priority.

Integrity of data is ensured by employing self-certifying data using public key signatures. Once the data is obtained, clients can verify both data integrity and the authenticity of the sender without using any additional network resources. This concept is applied to node ID certificates, the content distributed on the overlay, as well as all information published by the supernodes.

The deployment of a full PKI is out of the scope of this dissertation. However, the authenticity of the overlay information sent by the supernodes is of importance. The overlay information contains the global network view and content flow data, rather than plain content. The overlay information is generated at the supernodes and its integrity must be protected. The most convenient way to ensure integrity of overlay information is to allow supernodes to include digital signatures in those messages. In turn, this involves assigning private-public key pairs to supernodes. The signatures are verified at regular nodes. The root certificate used to issue supernode certificates will be distributed to regular nodes along with the client software. This architecture has the benefit of using self-certifying data for the

overlay information.

3.2.3 Application Security

Another security issue is exposing the P2P client application to the Internet. Connectivity is crucial for a P2P application, so it will employ NAT traversal mechanisms to improve connectivity. However, opening ports on a NAT device is considered a security threat by many network administrators. A common concern with P2P clients is that eventual application bugs and vulnerabilities are exposed to remote attackers, allowing the spread of worms and trojans.

Therefore, extreme care has been taken to avoid programming errors. The Contentifier application is developed in Python, a high-level programming language. The choice of a high-level language largely eliminates the possibility of memory leaks and buffer overflows, that account for many vulnerabilities in applications developed in low-level languages. Another advantage of high-level languages is that the application code is relatively short. Less code means less bugs. The Python syntax and the simplicity of the language produces readable code, which in turn makes code reviews easier.

Twisted's concurrency model, specifically the lack of threads in general networking tasks reduces concurrent programming pitfalls and difficulties related to accessing shared resources, such as race conditions and deadlocks.

Secure programming guidelines have been followed throughout the development. The use of encapsulation, clean interface design, extensive error checking and exception handling has been taken into account. Sensible limits (on input data, amount of network resources used, timeouts, etc.) are enforced.

Furthermore, Test-Driven Development (TDD) has been employed to spot and isolate programming errors early in the development cycle. Refactorings have been made to keep the code simple and clean. A subversion code repository was used to keep track of changes. Documentation has been written and maintained literary along with the code (docstrings). Logging is used throughout the application.

3.3 Interfaces

3.3.1 Middleware-Application Interface

The application extends the middleware classes `RingProtocol` and `RingNode` with application-specific protocols and shared state and behaviour. The ring protocol provides message sending and receiving trough an open session. The `RingNode` provides facilities for sending messages to specific peers and implementing application-specific behaviour. Figure 3.4 gives an overview of the services provided by the two classes and the details are described in the next chapter.

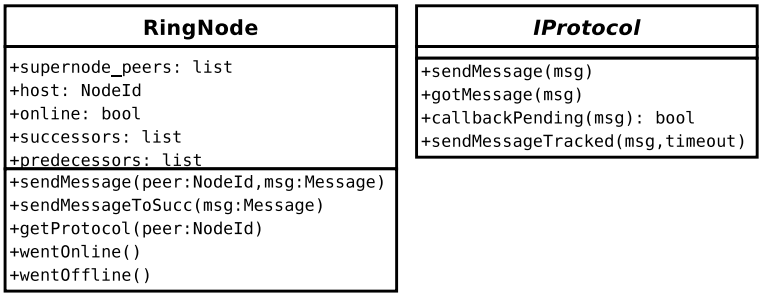

Figure 3.4: Middleware interfaces

3.3.2 Network-Frontend Interface

The network component provides the frontend with upcalls on network events. The frontend initiates downcalls to control the network component.

The interfaces between the network component and the frontend are consciously kept as simple as possible. The basic structure of the interfaces is described here. A pair of interfaces define how the two components interact.[13]

The network interface `INetwork`, as seen by the frontend:

start() Starts the networking module and event loop. Connects to the overlay network.

[13]The Zope Component Architecture [33] was used to define the interfaces.

stop() Shuts down the networking module and event loop, closing all open network connections. Stops the ring protocol.

sendContent(file, metadata) Attempts to insert new content into the overlay. The frontend invokes this downcall when the user has created new content.

The upcalls are defined in the frontend interface. The frontend interface `IFrontend`, as seen by the network:

onlineStatusChanged(status) Notifies the frontend about the online status. The middleware invokes this upcall when joining a ring has succeeded and after leaving a ring or network failure. The upcall is also used to indicate that the node seems to be firewalled and attempts for automatic NAT traversal have failed.

networkTopologyChanged(topology) Notifies the frontend about a change in the network topology. This upcall is invoked when information about the global network view and content flow arrives. The information includes geo-location of peers and content. This information is displayed to the user.

gotContent(file, metadata) Notifies the fronted about the arrival of new content that has been downloaded. The frontend presents the content to the user.

Chapter 4

Implementation

This chapter gives an overview of the development process and presents the tools used in the implementation. Following that, a detailed report on the implementation and software architecture is given.

4.1 Development Process

Team. The Global Contentifier project is being realised in a team effort of international collaboration. Thom Kubli has created the core concepts of the Global Contentifier for an arts project. He defined the main technical requirements and manages the project. The author of this dissertation designed and implemented the network component of the software for realising the Global Contentifier. Uli Fouquet develops the GUI frontend component of the software and provides the necessary infrastructure for the development.

Meetings. Initially the team has met up in person to discuss the requirements and possibilities of addressing them. Based on the initial discussions the system design was proposed. Later on, the team had regular online meetings to evaluate the progress continuously by testing the implementation, refining interfaces and specifying short term goals for the iterative development. The short feedback cycle has proved to be very effective.

Development. The iterative development was also employed on a smaller scale. In particular the software was developed using test-driven development to keep the feedback cycle short on the code level as well. Unit tests were written as a way of low-level design and documentation and were maintained throughout refactorings. A subversion code repository was used to keep track of changes and progress.

4.2 Implementation Choices

4.2.1 Python

Python [34] was chosen as a programming language for development. Reasons for this decision are detailed here.

Python is a high-level, multi-paradigm, dynamically typed programming language. It encourages agile development, as it strives to lessen programming effort by its simple and powerful syntax. Python code tends to be short and readable, minimising the possibility of bugs and maximising maintainability. These properties make Python an excellent tool for prototyping. Python is largely platform independent, moreover, the Python interpreter is shipped with most major operating systems out of the box except Windows and implementations for the JVM and the CLR exist[1].

Python was first published by Guido van Rossum in 1991. Currently it is actively developed in an open source effort and has a strong user base. Python has an extensive standard library and many third party modules.

The concept of docstrings is a unique language feature that allows the embedding of documentation into the code, which in turn improves maintainability[2].

4.2.2 Twisted

Twisted [35, 36] is a non-blocking, asynchronous and event-driven network programming framework for Python. It is powerful, flexible and highly modular.

[1] Jython and IronPython respectively.

[2] Docstrings are not only useful for generating documentation, but can be used to retrieve the documentation of modules, classes and methods at runtime.

Twisted supports a wide variety of network protocols (SSH, HTTP, Jabber, SIP, etc.) and transports (TCP, SSL, IP multicast, files, ...) out of the box and can be extended with more. Protocols and transports are completely separated and modular. While Twisted is a high-level framework, its low-level workings are not obscured and accessible if necessary. While it is platform-independent, it exploits platform-specific features like high performance multiplexing for non-blocking IO.

Twisted is also open source software. The development process used by the Twisted team is very effective. Test driven development is employed and code reviews are an integral part of the process. The user and developer communities are vibrant and helpful.

Deferreds. Deferreds are a core idea of Twisted. A deferred can be returned by a function, indicating a possibly long-running process with no immediate result, but promising a result in the future. Callbacks and errbacks can be attached to a deferred, which will be executed when the result is ready and when the operation has failed respectively. Deferreds can be chained and multiple callbacks (errbacks) can be added to a single deferred. The mechanism of callbacks and errbacks can be compared with flow control through signal and exception handling. Deferreds allow for writing of asynchronous code that looks very similar to the synchronous counterpart.

Tightly coupled to deferreds is the concept of cooperative concurrency[3]. Instead of blocking until results are available, twisted functions return a deferred immediately and delegate further work into the callback chain of that deferred. Larger tasks are refactored into smaller steps. This concurrency model is similar to coroutines and closures and combined with non-blocking IO largely eliminates the need for threads. The lack of threads in turn improves performance and mitigates common concurrent programming pitfalls[4].

Threads can be still used if there is no convenient way to split up a long running operation in the case of an external program or module. For such cases Twisted provides classes that manage a task queue and a thread pool. Events are

[3]Also green threads or microthreads.

[4]At this point it is worth mentioning that Python threads are somewhat flawed from a scalability perspective, since the Global Interpreter Lock forces atomicity of all statements and significant context switching penalties apply.

generated in the main thread to notify about task completion or failure.

Modular Design. Twisted applications are conventionally structured in a modular fashion. The network connections are represented by objects inheriting from the `Protocol` class. The protocol provides network event handlers that are called when the connection is made, lost, or data is received. The protocol objects are instantiated by a `Factory` subclass. The factory is also used to share state between the protocol instances. The reactor runs the event loop, connects the factory to a network transport[5], polls sockets and dispatches data flow to and from the protocol instances.

Twisted applications define the protocol, customise the factory, and control the reactor.

An example can clarify the benefits of flexibility and the high degree of modularity. An application developer having written a simple HTTP client might want to enhance the program to support encryption. The developer might also want to add a non-standard extension to the HTTP client that closes the connection after receiving a given number of bytes. A third addition might be to throttle the HTTP connection to preserve bandwidth for more important tasks. Each of these changes will affect a single piece of code: the code controlling the reactor, the protocol class and the factory class respectively. In particular it would involve replacing the TCP transport with an SSL transport, adjusting the event handler responsible for receiving data, and using a Twisted policy mixin to extend the factory class to support throttling.

Testing. Twisted also includes Trial, an extension of the Python unit testing framework. Trial makes it convenient to test protocols, networking code, code depending on timing and asynchronous event-driven code in general. Writing unit tests for these types of code is traditionally hard; Trial relies on the modular Twisted architecture to provide convenient ways of unit testing.

[5]Common network transports are UDP, TCP, and SSL.

4.3 Implementation Details

4.3.1 Overview

Figure 4.1 illustrates the three components of the P2P client software and the basic class structures. The *middleware* implements the ring protocol, the *application* provides the application layer protocols. The *frontend* produces and consumes content and controls the application.

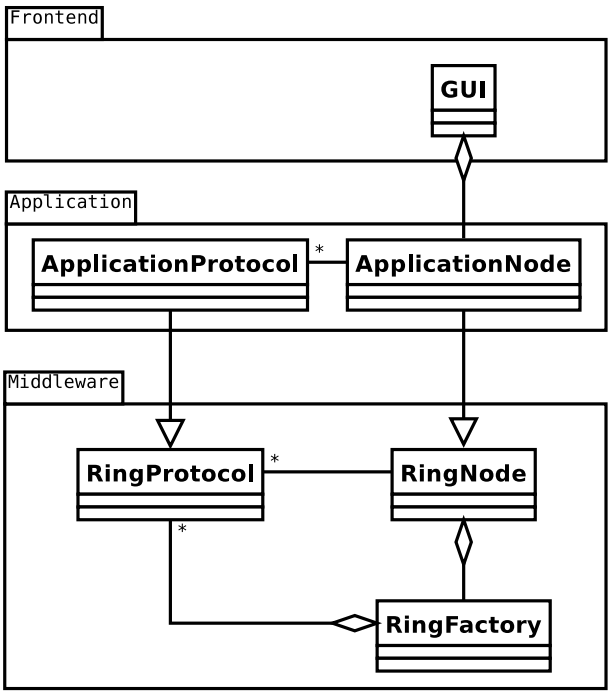

Figure 4.1: Components of the P2P client software

Middleware. The `RingProtocol` class implements the syntax and semantics of the ring protocol. The `RingNode` class shares state between and controls the protocol instances. The RingFactory class creates protocol instances and binds them to incoming and outgoing network connections.

Application. The application using the middleware extends the `RingProtocol` class with application layer message handlers. The application also extends the

`RingNode` class by adding application-specific shared state and behaviour in the `ApplicationNode` class.

Frontend. The frontend instantiates the `ApplicationNode` class and uses it to control the underlying components by initiating downcalls and to receive events via upcalls.

4.3.2 Protocol Stack

The communication protocol is implemented in a layered architecture. Each protocol layer provides services and abstractions for and encapsulates data of the next layer above. Each layer is implemented in a separate class. The lowest protocol layer subclasses the Twisted `Protocol` class and the following layers inherit from each other respectively. The protocol stack is shown in Table 4.1.

Layer	Protocol
4	Application Protocols
3	RingProtocol
2	MessageDispatcher
1	MessageSerializer

Table 4.1: Protocol Stack

Details of the responsibilities of the different layers are described here.

MessageSerializer. The lowest protocol provides message encoding. A text-based header is optionally followed by the binary payload. The main header includes the size and checksum[6] of the payload and message-specific headers. The payload size is important for synchronisation as it indicates how much binary data is to be read before the next message header. The message-specific headers contain structured data and are encoded in the JSON format[7]. The `MessageSerializer` class

[6]The current implementation supports CRC-32 checksums and SHA-384 hashes for integrity checking.

[7]The JavaScript Object Notation [37] is a lightweight data interchange format. It is easy for humans and machines to read and write. It can represent nested structured data (ordered lists and collections of name/value pairs) and primitive types (numbers, Unicode strings, boolean and null values).

```
1 180 -503725105
2 {"from":{"ip":"127.0.0.1","id":"ONE","port":1404},"succ":[{"ip":"
    127.0.0.1","id":"TWO","port":3333}],"type":"HelloReply","to":["
    127.0.0.1",3641]}
```

Listing 4.1: `HelloReply` message serialised to wire format

is responsible for splitting (merging) messages from (to) the underlying stream-based transport, marshalling of `Message` objects into the wire format, as well as integrity and error checking.

Listing 4.1 shows a serialised message. The main header in the first line shows the message size and a CRC checksum. The message-specific header (line 2) reveals the message type, source and destination and contains a structured successor list. The lack of the third line indicates that there is no actual payload apart from the headers.

It is obvious that the encoding is not terse. The implementation allows for the encoding to be exchanged with a binary encoding[8] if the overhead is deemed too large.

MessageDispatcher. The `MessageDispatcher` class is responsible for calling the appropriate message handler method based on the type of the incoming message. The dispatcher makes use of Python reflection to determine the validity of a message type and safely call the corresponding message handler, eliminating the need of maintaining mapping code for each message type. `MessageDispatcher` also provides methods for sending messages. It uses the Twisted `TimeoutMixin` to clean up and close unused connections.

This layer also encapsulates the request-response matching pattern. It is fairly common to send a message and wait for a specific response. It is achieved by sending a tracking ID in the message header and automatically filtering incoming messages to find the response with the same ID. Boilerplate code is abstracted away and thus the readability of higher-level code is improved. The developer of higher level code uses simple control structures. Consider the non-blocking code that sends a ping message and waits for a response or timeout, shown in Listing 4.2.

[8]For example *bencode* is a binary encoding that supports structured data. It is used by the BitTorrent protocol.

```
1  @inlineCallbacks
2  def ping(proto, timeout=10):
3    msg = messages.Ping()
4    try:
5      # this might take a while:
6      response = yield proto.sendMessageTracked(msg, timeout)
7    except TimeoutError:
8      print 'request timed out'
9    else:
10     print 'response is', response
```

Listing 4.2: Example of request-response pattern

The `proto` parameter is a `MessageDispatcher` protocol instance, that represents an open connection. The `ping` method sends a `Ping` message by invoking the `sendMessageTracked()` method on the protocol object. After the invocation it yields control[9] immediately. Thus the rest of the program can continue its usual business until the response arrives or a timeout occurs. When the response to the `Ping` message is received or after a timeout of ten seconds the control is given back to the method and the execution continues where it left off: at line 6 with assigning the `response` variable and processing the outcome of the ping request.

This pattern encourages the use of a high-level mental model for writing non-blocking request-response code with optional support for timeouts.

RingProtocol. This layer implements the ring protocol discussed in detail in Section 3.1.4. The implementation includes message handlers for the Hello protocol, the join operation and the failure detection and recovery.

Figure 4.2 shows a state machine describing the behaviour of the ring protocol. The state transitions are triggered by messages being sent and received. The protocol starts in the offline state. In the join state the negotiation with the neighbours is in progress. The online state is entered when the join procedure was successful and is left when the successor fails. The recovery from the failure of one or more successors is attempted in the repair state.

[9]The Python `yield` expression is used to implement coroutines [38]. The coroutine observes the behaviour of the `yield` expression as a simple function call: parameters are passed, it "blocks" for a while, then a value is returned or an exception is thrown. Here a deferred is passed as the parameter and a message object is returned. In the background the `inlineCallbacks` decorator adds a callback to the deferred that returns control to the coroutine after the deferred fires.

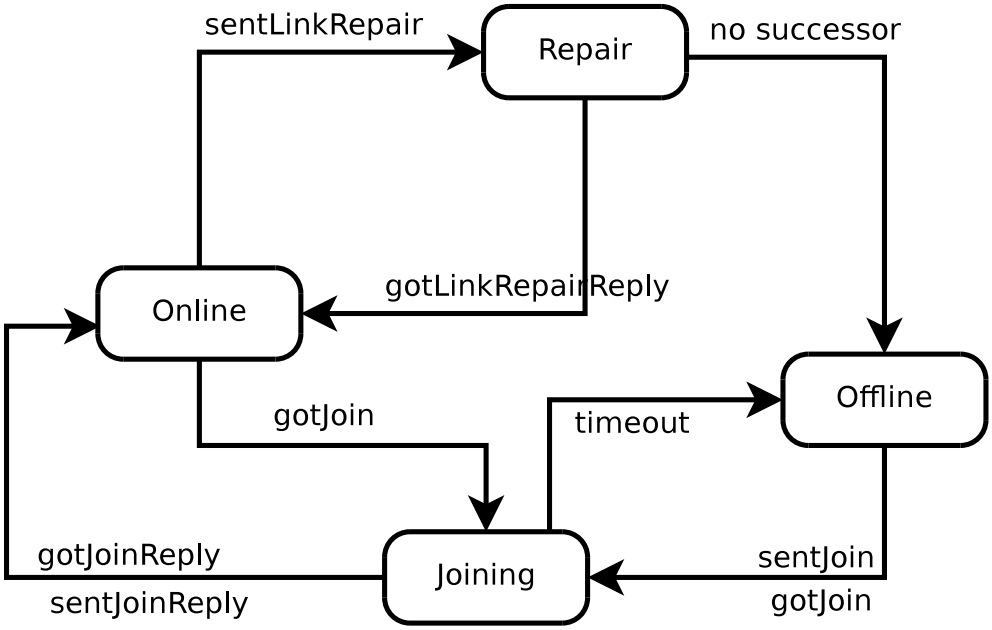

Figure 4.2: State machine describing the ring protocol

ApplicationProtocol. The application protocol provides message handlers for application level messages. In the content protocol these are related to content announcement, upload and download, as well as metadata approval.

When inserting new content, the node requests a permission to broadcast from a supernode, that eventually signs the metadata. Given permission to broadcast, the node announces the new content to its successor on the outer ring. The content announcement includes the metadata. The successor verifies the metadata and requests the content if it has no copy of the content in question. The content is split up in equal pieces and the individual pieces are transmitted upon request. After a node completes downloading the content, it verifies the content hash included in the metadata, then initiates a `gotContent()` upcall and announces the new content to its successor to ensure data flow on the ring.

The sequence of messages exchanged during content insertion is illustrated in Figure 4.3.

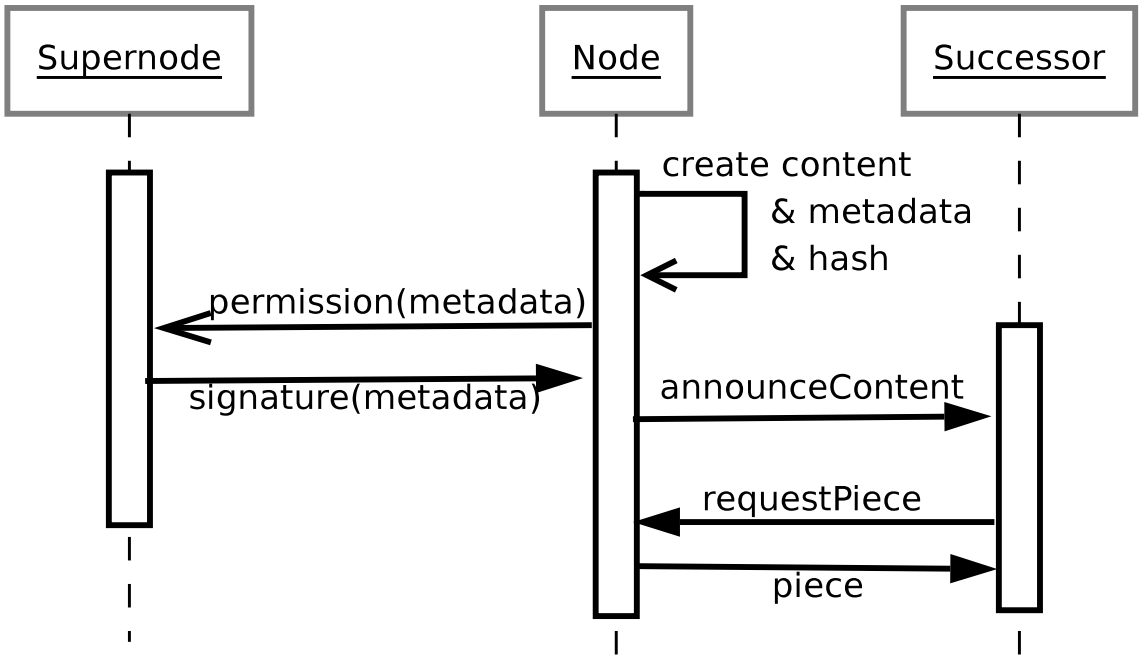

Figure 4.3: Content Protocol: messages exchanged during content insertion

4.3.3 Shared state

While the ring and content protocols are responsible for communicating with peers and define the sequence of exchanged messages, the classes inheriting from `RingNode` share state between the protocol instances (network connections), keep track of them and support the ring protocol with management and maintenance tasks.

The shared state includes the neighbourhood set, the own node ID, online status, content metadata, known peers including bootstrapping peers as well as a list of known supernodes. The maintenance tasks realised include the heartbeat, control of the registration and node join operations and control of content flow. Basic routing facilities are also provided.

Ring. The `RingNode` class implements state storage structures and management tasks specific to the ring protocol. Its `heartbeat` method is called periodically. It drives the hello protocol by periodically instructing the ring protocol to send `Hello`

messages and monitoring `HelloReply` messages, detecting eventual failure of the successor and initiating the ring repair. While the node is offline, the heartbeat is responsible for establishing connectivity using the `connect` method.

The `connect` method initiates the registration process and the ring join process. It monitors the outcome of the operations and takes the appropriate actions. These include making the appropriate upcalls to notify the frontend of changes in online state as well as trying other (super-)nodes while backing off exponentially in the case of failure.

Content. The `ContentNode` class implements the application layer part of the node state. Its main responsibilities are the storage of metadata and the management of uploads and downloads. The announced and requested contents are added to the download and upload queues, respectively. The `download` method schedules downloads and controls the content protocol to request the appropriate pieces. The `ContentNode` class cooperates with the `filesharing` module to accomplish these tasks.

The `filesharing` module encapsulates classes for storing content metadata and managing the details of file transfer. The content is never kept in memory, instead it is written to (read from) the hard disk piece by piece. The current implementation only supports ordered sequential download of the content pieces, which is sufficient at this time. The success of the file transfer is verified after the download is complete by calculating the hash of the content and comparing it to the one stored in the metadata.

4.3.4 Messages

The `messages` module encapsulates the `Message` base class and all valid messages. It provides functionality for message marshalling and demarshalling. All messages have to inherit from the `Message` base class, as only these will be accepted by the message passing protocols. Additional message types include `ApplicationMessage`, `RingMessage` and `TrackedMessage`. The latter mixin class provides functionality for the request-reply pattern (described in Section 4.3.2) and is incorporated using multiple inheritance. Figure 6.1 shows the class diagram of the `messages` module.

```
>>> import messages
>>> m = messages.Message(spam='eggs', foo='bar')
>>> 'spam' in m
True
>>> m.header.get('spam')    # traditional behaviour
'eggs'
>>> m['spam']    # dict behaviour
'eggs'
>>> m.spam    # object behaviour
'eggs'
```

Listing 4.3: Accessing message headers in the `Message` class

The `Message` class behaves like a `dict`[10] to make it convenient to get and set header attributes. Note that the class could be sweetened with even more syntactic sugar by implementing object-like behaviour[11]. The interactive Python shell in Listing 4.3 illustrates the difference.

4.3.5 Node Join

New nodes that want to participate in the overlay initially have to go through two steps. First the new node registers with a supernode and then it joins the ring.

Registration process. Before joining the overlay a new node has to register with a supernode. The reasons for this step are twofold. The supernode participates in the generation of the node ID and supplies bootstrapping information in the form of node IDs of geographically nearby nodes. The overall workings of the registration process have been described in Section 3.1.5 and are discussed here in detail.

Before using resources of the supernode, the new node has to provide proof-of-work by solving a crypto puzzle based on the address of the supernode and the port the node is listening on. After verifying the crypto puzzle, the supernode checks the connectivity to the joining node and asks another supernode to do so as well. If the joining node cannot be reached from the Internet, it has to resolve this issue before participating in the network.

[10] The Python `dict` type is a dictionary of unordered key-value pairs. Other languages use the term "hash table".

[11] This behaviour could be implemented with ease using metaprogramming.

Then the supernode geo-locates the joining node based on its IP address. A unique node ID is assigned to the joining node and it is sent along with bootstrapping information: a list of node IDs of peers in the geological area of the joining node. The node ID also contains the geo-location and public IP address of the joining node and is signed by the supernode, so that it can be verified by third parties. Table 4.2 describes the data stored in the node ID data structure.

The supernode stores the node ID of registered nodes. Node IDs are valid for a single session and expire after a configurable amount of time. The stored node IDs are exchanged with among supernodes and used to aid new nodes in finding geographically close nodes in the overlay.

Attribute	Description
id	A globally unique identifier of the node
ip	Public IP address
port	Network port the client is listening on
location	Geo-location of the node
public_key	Public key (only set for supernodes)
expire	Timestamp when the node ID expires
signature	Digital signature of a supernode

Table 4.2: Data stored as part of the node ID

Node Join. Having assigned a node ID the node can start the process of joining the ring. Again the joining node has to solve a crypto puzzle and attempt to join the ring. The node receiving a join request verifies the crypto puzzle and the node ID and updates its current neighbours about the joining node. After being accepted, the joining node runs the hello protocol and participates in the overlay.

The node join process involves three nodes: the joining node and its neighbours-to-be. The successor lists are distributed shortly after the join process using the hello protocol. Ring continuity is maintained throughout the join process unless one of the nodes decides to cancel the operation.

The sequence diagram in Figure 4.4 illustrates the exchanged messages. The lifelines indicate the online status of the nodes. The actual network topology is depicted as well. Nodes A and B are initially in a ring. Node C joins in between them, making B its successor and A its predecessor.

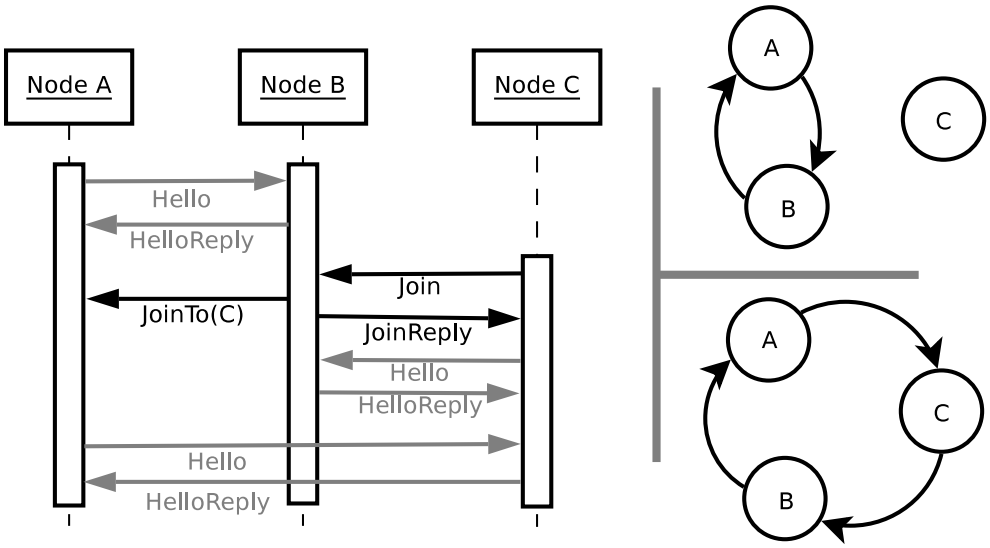

Figure 4.4: Messages exchanged during the join operation. The diagrams on the right hand side illustrate the ring topology before and after the join operation. The messages highlighted in grey colour are not part of the join operation.

4.3.6 NAT traversal

NAT traversal is crucial for any P2P application targeting home users (see Section 3.1.2). It has been decided that TCP hole punching is too complex and lacks sufficient documentation and open source implementations.

The current implementation makes use of the UPnP protocol to detect and control home routers. The Nattraverso library [39] realises the necessary functionality as an API. It is based on the Twisted framework and is licensed under the LGPL[12].

The `PortMapper` class is part of the middleware and provides convenient management of port mappings on top of Nattraverso. It enumerates UPnP devices on the LAN and keeps track of active mappings to enable the addition, removal and monitoring of the port mappings. The middleware attempts to enable port forwarding at startup if any suitable devices are found. The results are communicated to the application layer via the upcall `onlineStatusChanged()`. The port mapping is removed on cleanup.

[12]GNU Lesser General Public License.

```
2007-09-13 01:55:41,880 cntfr.ring CRITICAL: connect: RegisterError (we are offline). retry in 5 sec (ring.py:330)
2007-09-13 01:55:47,320 cntfr.ring DEBUG: trying to bootstrap from: [['127.0.0.1', 3333]] (ring.py:307)
2007-09-13 01:55:47,320 cntfr.ring DEBUG: trying to join ring (ring.py:408)
2007-09-13 01:55:47,322 cntfr.ringproto DEBUG: connection established with peer ['127.0.0.1', 3333] (ring_protocol.py:44)
2007-09-13 01:55:47,325 cntfr.ringproto DEBUG:      -- sending message (Join) to ['127.0.0.1', 3333] (ring_protocol.py:161)
2007-09-13 01:55:47,333 cntfr.ringproto DEBUG: peer isolated (ring_protocol.py:335)
2007-09-13 01:55:47,340 cntfr.ring INFO: 1 NEW SUCCESSORS (ring.py:200)
2007-09-13 01:55:47,341 cntfr.ring DEBUG: join ok. (ring.py:453)
```

Listing 4.4: Log file excerpt

4.3.7 Logging

Extensive logging has been developed along with the middleware and the application. Even tough logging is not a main requirement, it certainly is useful for testing, analysing and debugging distributed systems. The logs of the nodes in an overlay test run can be collected and combined. The combined logs allow for reasoning about the overlay behaviour or track down specific problems.

The Python logging facility is utilised, as it provides the necessary features like log levels, log file rotation, as well as multiple destinations and formatters. The logging format is verbose: the timestamp, log level and the exact location of the logging call itself are added automatically. Listing 4.4 shows a log excerpt of a successful join operation.

4.3.8 Frontend

The frontend handles user interaction, produces content to be broadcast and consumes content arriving on the ring. The frontend also presents the global view and content flow information to the user.

Two frontends have been developed for the Contentifier application. A console-based user interface and a GUI. The console client has been developed along with the middleware and application. The GUI is currently being developed by

team member Uli Fouquet and is in the phase of integration.

GUI frontend. The graphical user interface frontend is not part of this dissertation, but an important part of the Global Contentifier project. A short overview is given here. The GUI provides audio recording and playback, as well as visualisation of an interactive world map showing the overlay network topology and the content flow therein. A screenshot of the current version of the GTK-based GUI is shown in Figure 4.5.

The Speex codec [40] is be used to encode audio data. Speex was designed for encoding high quality human speech at a low bitrate. The codec is patent-free and open source implementations exist.

The global network view visualisation is intended to be interactive to provide the user with the feeling of being "a part of the whole". Thus the visual representation will feature near-realtime updates of node joins and leaves and content flow progress in the overlay network. The visualisation will appear as unique and artistic animated infographics.

Console frontend. The console client is kept as simple as possible and was developed solely for testing and debugging purposes. The client has a built-in Python shell[13] to allow for debugging the running node with the full power of the language. It also provides a set of commands to use chat, insert content and display various statistics and node state. Furthermore, simulation of failure is implemented for testing; it supports single failure with or without recovery at a later time and has a random failure mode.

The console client can be configured with an extensive set of command line parameters and the configuration can also be changed at runtime. The client filters logging based on the current debug level and displays the desired events.

Listing 4.5 shows an example session of the console client. The numbers at the beginning of lines indicate the debug level of log messages. The lines starting with triple greater-than signs are displayed on arrival of upcalls.

[13]The user input loop is run in a separate thread in order to avoid blocking of the reactor. Entered commands are dispatched to and processed in the main thread. Python lacks asynchronous console input and Twisted only provides remote telnet and SSH shells for debugging.

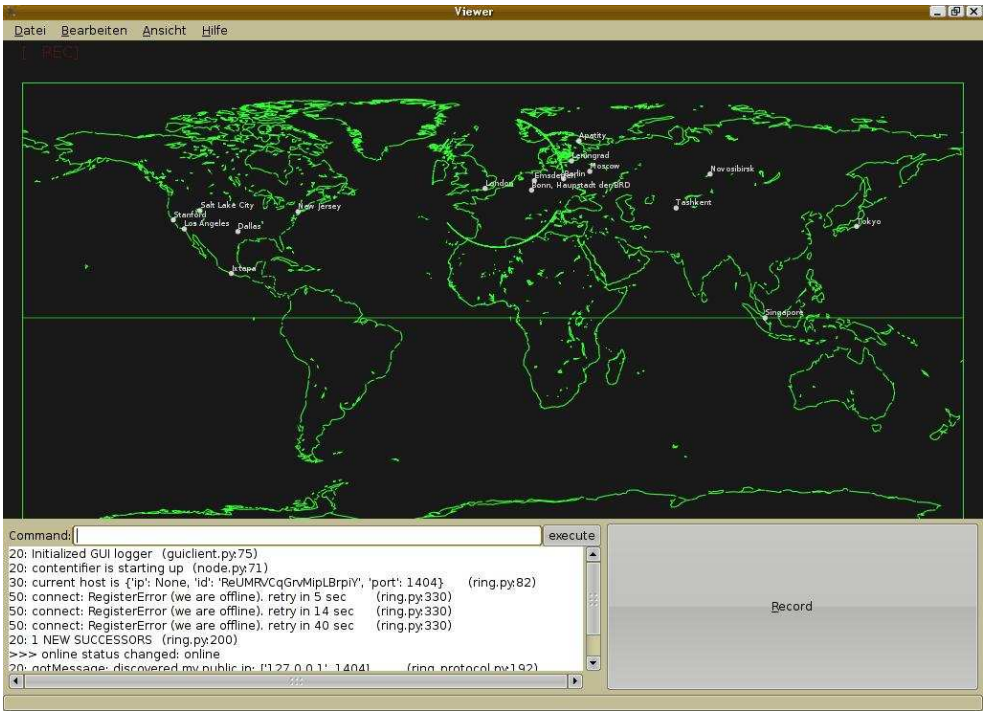

Figure 4.5: The GUI of the Contentifier application

```
$ ./contentifier.py --supernode localhost:1404 --port 2000 --id ONE
20: contentifier is starting up
30: current host is {'ip': None, 'id': 'ONE', 'port': 2000}
20: gotMessage: discovered my public ip: ['127.0.0.1', 2000]
20: 1 NEW SUCCESSORS
>>> online status changed: online
20: new content announced. downloading 218656 bytes...
50: content downloaded to downloads/bach.mp3  ...speed: 128.05 KB/sec
>>> got content. type "send downloads/bach.mp3" to forward it
50: last piece of content sent
```

Listing 4.5: Console frontend

Chapter 5

Evaluation

This chapter evaluates the research project. The evaluation aspects are twofold. First, basic performance metrics and messaging overheads of the middleware are quantified by evaluating the measurement results of a series of experiments. Second, the initial research aims and the defined requirements are compared to the actual achievements.

5.1 Measurements

Several tests and experiments have been conducted during the development and the results were the basis for refinements in design and implementation. The team members provided help resources in testing the functionality and stability of the system.

In contrast, this section presents scientific experiments made on the final system. Or more accurately, the latest version of the system.

Using sophisticated simulation tools would extend the significance of the measurements. On the other hand, simulations require careful planning and considerable effort in adapting the protocols to conform with the simulation framework in use. Instead of simulation, measurements on a set of live nodes were performed.

5.1.1 Experimental Setup

The measurements were taken on a single computer, a Dell Latitude D400 notebook. The machine has an Intel Pentium M 1.60 GHz processor and 512 MB of RAM and runs Ubuntu Linux 7.04 operating system with a 2.6 series kernel. Python 2.5.1 and Twisted 2.5 were installed on the system. Swapping was turned off and the Contentifier application logging was redirected to /dev/null to eliminate inaccuracies related to disk access.

Some measurements were conducted on a network of up to 50 nodes. This involves running 50 instances of the Python interpreter in 50 separate processes. All nodes share the resources of a single computer and significant penalties occur from e.g. frequent context switching. While the resources are low, the nodes only have to communicate on the loopback network interface which behaves like a network connection of unrealistically high speed. Therefore, the accuracy of the measurements can be disputed.

A single supernode was used in the experiments to evaluate the extent of bottlenecks building up at the supernode.

A simple script called `noderunner` was developed to start the desired number of nodes with a precise timing. The program takes care of supplying command line arguments to the Contentifier console client instances, such as setting the network address of the supernode, configuring a port and controlling logging output for each node. Each node is run in a separate `xterm` terminal to visually verify the progress. The overhead of running the terminals is a quantité négligeable.

5.1.2 Round Trip Time

In this experiment the round trip time of a short message traversing the entire outer ring was measured. A round trip message and corresponding message handlers were implemented at the application layer for this purpose. One node periodically sends round trip messages along the ring and measures the time it takes for the message to arrive back. A hop counter inside the message is incremented at each node that forwards the message. Thus the round trip message not only measures the round trip time, but also determines the current number of nodes in the ring.

The experiment started with a stable ring of two nodes and up to 50 nodes

were started successively with a two second delay each. The round trip times and the according number of nodes in the ring were logged and processed later. The experiment was repeated several times with similar results.

The network diameter grows as more nodes are added. Figure 5.1 shows the expected linear increase of the round trip time. Starting from 10 ms for two nodes the round trip time eventually reaches 250 ms in a ring of 50 nodes. The sparse outliers are attributed to context switching.

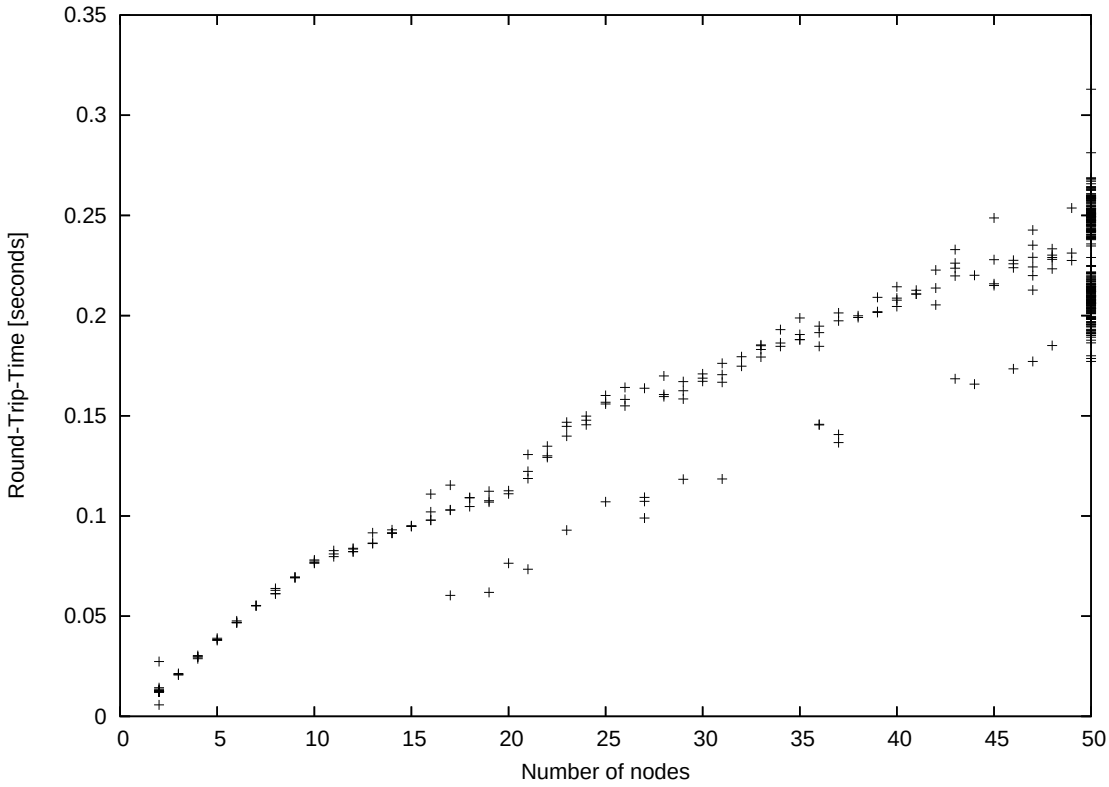

Figure 5.1: Round trip time and ring size

5.1.3 Messaging Overheads

An experiment has been conducted to measure the overhead of control messages over the content messages. The experiment was designed to measure the number as well as the size of messages for the ring protocol and the application protocol.

Ring protocol messages in this experiment comprehend building the ring from scratch and ring maintenance during transmission. Application messages are solely content-related and include the content announcement, piece requests and piece replies.

The experiment was conducted on a network of two nodes, as the number of nodes does not affect the measured values. A file of size 218 KB was transmitted from one node to the other and back. That is 436 KB of payload was transmitted altogether.

Collection of message statistics is part of the network component, hence no additional implementation specific to this experiment was required. The message statistics were retrieved after the transmission using the debug console built into the console client.

Figure 5.2 shows the results. The experiment produced 11 ring protocol and 30 content protocol messages. The cumulative size of all ring protocol messages make up only 0.32 % of the whole size. That overhead is considered acceptable.

5.1.4 Time to converge

This experiment quantifies the time it takes for a given number of nodes to converge to a single ring. First a supernode is started. All other nodes are started simultaneously. The nodes register at the supernode and join the ring. Note that three nodes are involved in a join operation and each node is allowed to participate in a single join operation at any time. Nodes back off exponentially when the join operation fails.

The experiment was conducted using 10 to 50 nodes in steps of ten. The time to converge was determined by measuring the round trip time periodically several times a second. The round trip message also includes a hop counter which was used to identify the point in time when all nodes are accommodated in the ring. The round trip measurements were logged for later processing.

Figure 5.3 shows the results. For 10 nodes it takes less than two seconds to converge, while 50 nodes converged in 36 seconds. Given a distributed ring protocol this takes quite long.

The unexpected results of this measurement point to an implementation

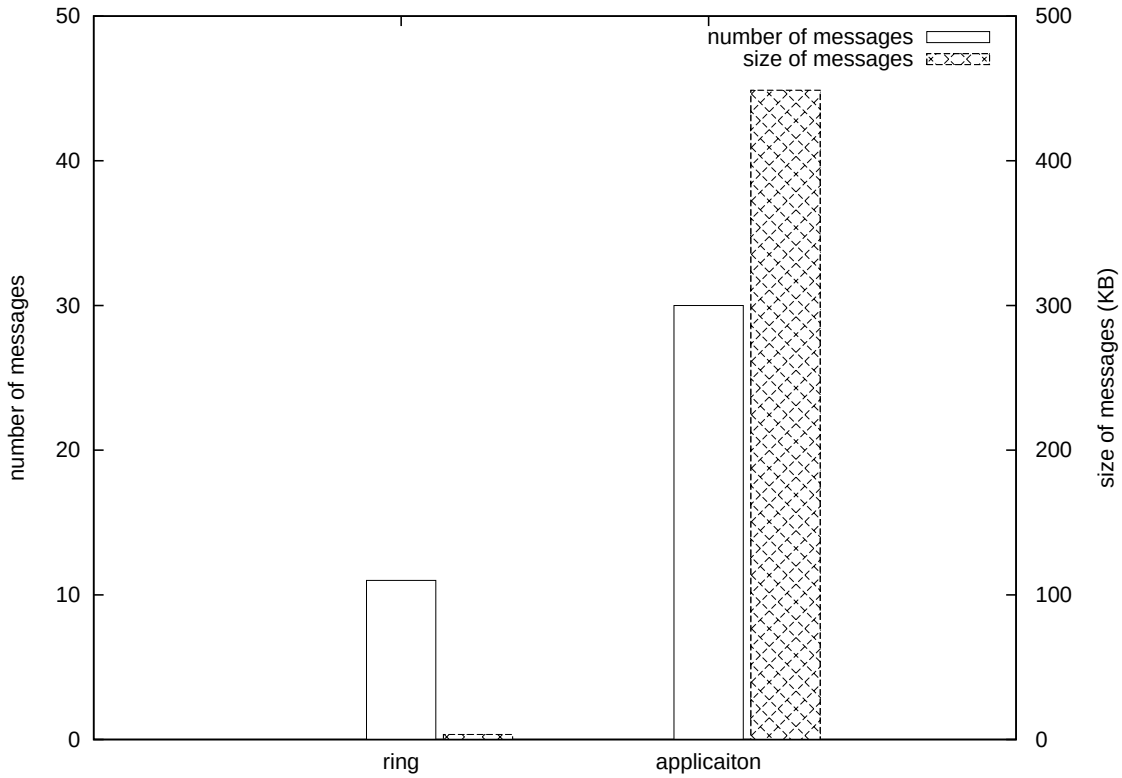

Figure 5.2: Ring protocol and content protocol messages

issue. The nodes are located at the same geographical position and the supernode returns the same bootstrap peer for each node. Hence a large number of join operations are attempted at a single node, most of which are ignored and the nodes time out, back off exponentially and try again. The back off time is deterministic, thus the nodes retry later simultaneously.

The problem could be addressed in multiple ways. One solution is to build randomness into the back off time. In this case nodes will initiate the join operation at different intervals and have a higher success rate. The other solution is to change the supernode algorithm to select a random bootstrap peer if there are multiple peers located at the same geographical position. In this case the nodes will initiate the join operation at different peers and can succeed in parallel.

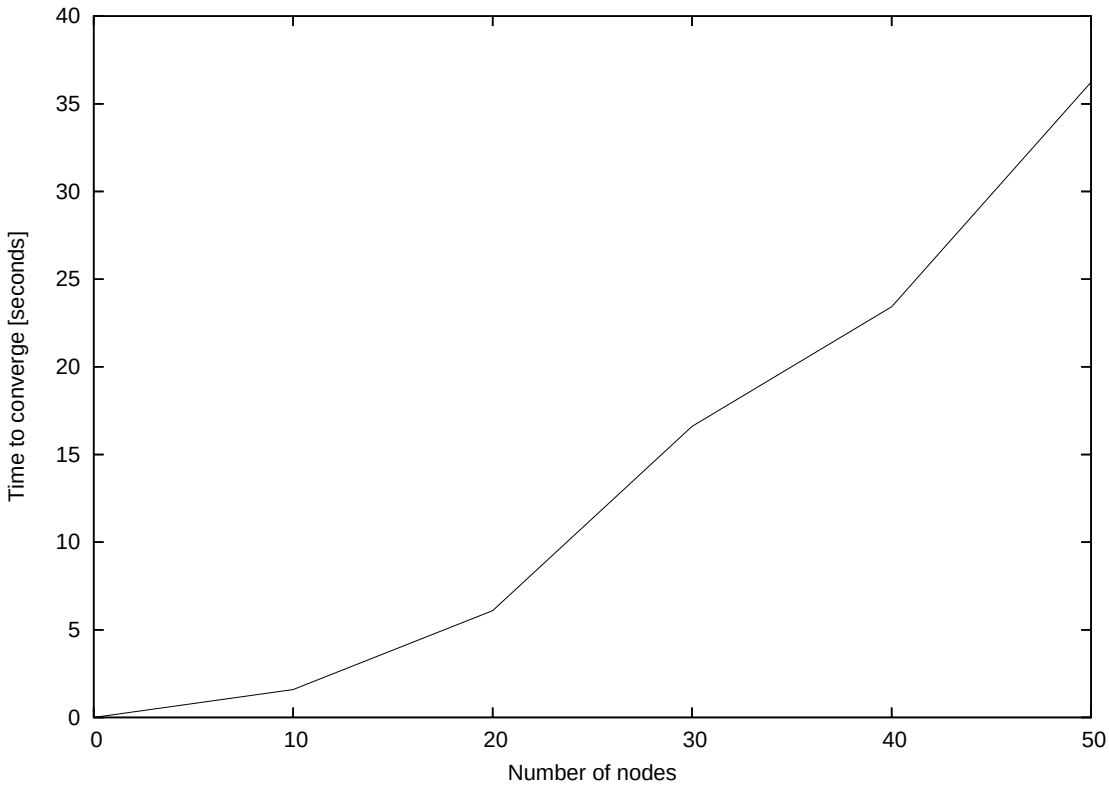

Figure 5.3: Time to converge to a ring

5.2 Goals and Achievements

The main goals of this research were outlined in Section 1.2. Most goals were eventually achieved. The state-of-the-art of ring-building overlays was surveyed and a ring protocol was adapted to the requirements of the middleware. The P2P middleware design and implementation is complete. Security threats have been considered and countermeasures have been designed and partially implemented. Notably, the digital signatures of supernode messages remain to be implemented. The networking component of the Global Contentifier application was designed and the implementation is nearly finished. In particular the content flow protocol was realised, while the implementation of the global view protocol is a work in progress. The enforcement of policies is in place, yet the dependency of the global view protocol needs to be satisfied. Section 6.1 details work that remains to be done.

Chapter 6

Conclusion

Directions for future work are discussed in this chapter, then the research is concluded.

6.1 Future Work

The future work is discussed in two sections. Ongoing work details the work that is already in progress, the design is done and in some cases the implementation has started. The outlined tasks are of highest priority for the Global Contentifier project and the work will be continued after the submission of this dissertation.

The next section discusses potential future work and identifies interesting directions for future research that is not scheduled yet for the near future as part of the Global Contentifier project.

Ongoing Work. Most importantly the implementation of the *global view protocol* has to be finished. The protocol entails reporting of network events to supernodes, aggregation of this information at the supernodes and distribution of the global view to the regular nodes. The protocol semantics and data structures in question have been agreed upon. Implementation of the new protocol layer remains to be done.

The view protocol outlined above depends on correct geo-location of the P2P nodes. The current geo-location routines use a web-based frontend to an open source database backend. However, concerns have been raised about the accuracy

of this method. It has been discussed that users should be able to locate themselves on the map, or alternatively, a traceroute-based method can be used in case the database fails. As correct geo-location of nodes is a crucial part of the system, more research needs to be conducted in this area.

Another important area for ongoing work is the implementation of digital signatures to allow for self-certifying data that originates from the supernodes. These include the node ID, the bootstrapping information, the metadata and the global view information. Of course the according verification processes have to be put in place as well.

P2P systems are relatively hard to evaluate and test. Clearly, more tests and measurements have to be conducted beyond the ones presented in Section 5.1. The best practice to evaluate P2P systems is to observe the behaviour of a real-life network deployed on a large scale. This remains yet to be done. An initial real-life deployment of medium scale is being organised at the time of this writing.

Potential Future Work. The policies for node joins and content insertion have to be fine-tuned and possibly other policies have to be developed based on experience from the initial real-life deployment. One possibility for a new policy is the requirement of minimum bandwidth that a node can contribute to the network. In a ring overlay the throughput of the network depend on the throughput at the slowest node. Even if the interventions are temporary, content flow needs to be ensured.

Further fine tuning and configuration of the P2P network could be realised to improve performance metrics. Possible values for fine tuning are various timeouts, the frequency of the ring protocol heartbeat, the number of successors that are being kept track of, the number of contents transmitted simultaneously at a single link, as well as the hashcash prices of different operations.

The source code of the middleware and the application will be released as free and open source software at some point. The software release is essential to allow for the development of other interesting projects and applications that are based on the ring-building middleware.

Making use of more open protocols and formats should be considered where possible. The potential benefits are numerous. It would be interesting to investigate the possibility of building the content protocol using a feed-based publish-subscribe

mechanism instead of a proprietary protocol.

Possibilities of interconnecting multiple rings at bridges. Brides are P2P nodes that are part of multiple rings and have the responsibility of mediating and exchanging content among the rings. Additional policies could control the way content is exchanged at bridges.

6.2 Conclusion

This section summarises the progress and achievements of this research and identifies the main contributions.

6.2.1 Summary

This paper is a report on the results and progress of a research that is part of an ongoing project. A summary is given here.

The state-of-the-art of ring-building overlays suitable for P2P systems and geography in the context of P2P applications are surveyed. The Global Contentifier media arts project is presented. The middleware and application requirements of the Global Contentifier are outlined and design issues are identified. Based on the survey and the requirements a distributed ring protocol is designed.

A supernode-based P2P middleware is designed and implemented. The middleware builds and maintains an overlay network of multiring topology wherein all peers are part of the outer ring and the inner ring provides low-latency communication for the supernodes. The middleware provides components and distributed services for the development of P2P applications. Security issues are considered and threats are addressed with countermeasures. An application for realising the Global Contentifier on top of the middleware is designed and implemented.

The results are evaluated by means of experimental measurements and contrasting the original goals with the actual achievements. Finally, future work is identified and the research is concluded.

6.2.2 Contributions

Ring Protocol. A distributed ring protocol based on VRing is proposed. The ring protocol is suitable for P2P systems and can tolerate a configurable amount of failure. It also eliminates limitations of the VRing protocol by adapting it to the requirements of the middleware.

Middleware. A ring-building, supernode-based middleware was designed and implemented. The middleware builds and maintains a P2P overlay network featuring the novel concept of serial content flow. The designed protocols take the geographical location of the nodes into account. Current evidence suggests that the middleware is in a way the first of its kind.

The middleware provides a platform for building interesting applications. Hence, the open-sourced middleware will contribute to the P2P research and the Twisted communities and hopefully inspire more research and development and also serve educational purposes.

Application. An application for realising the core concepts of the Global Contentifier was designed and partially implemented. The innovation of the application is the aggregation of a global view of the overlay network topology and the content flow therein. The application is not only of academic interest, but also contributes to a media arts project by enabling the novel interactive audio sculpture of the Global Contentifier.

Appendix

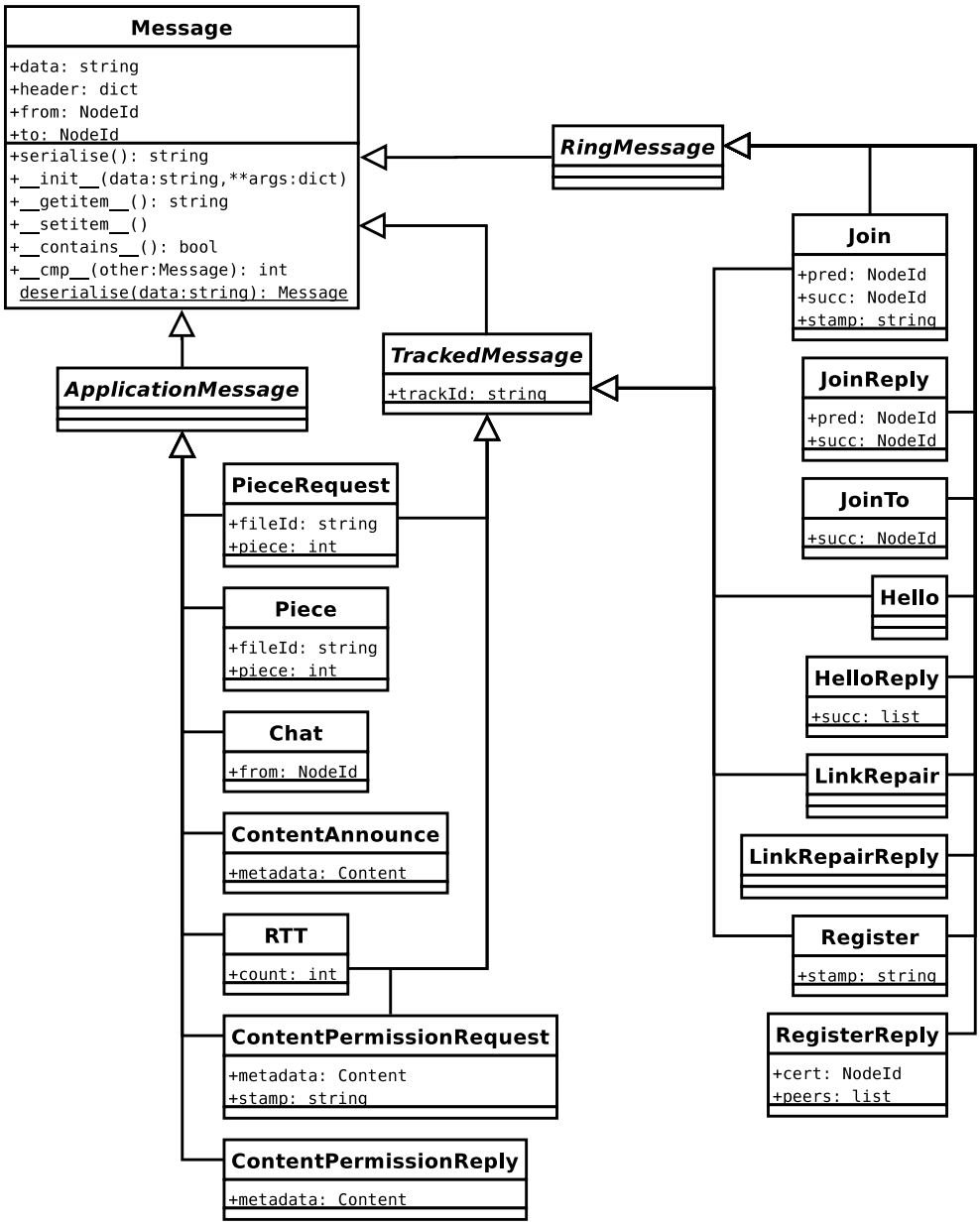

Figure 6.1: Class diagram of the `messages` module

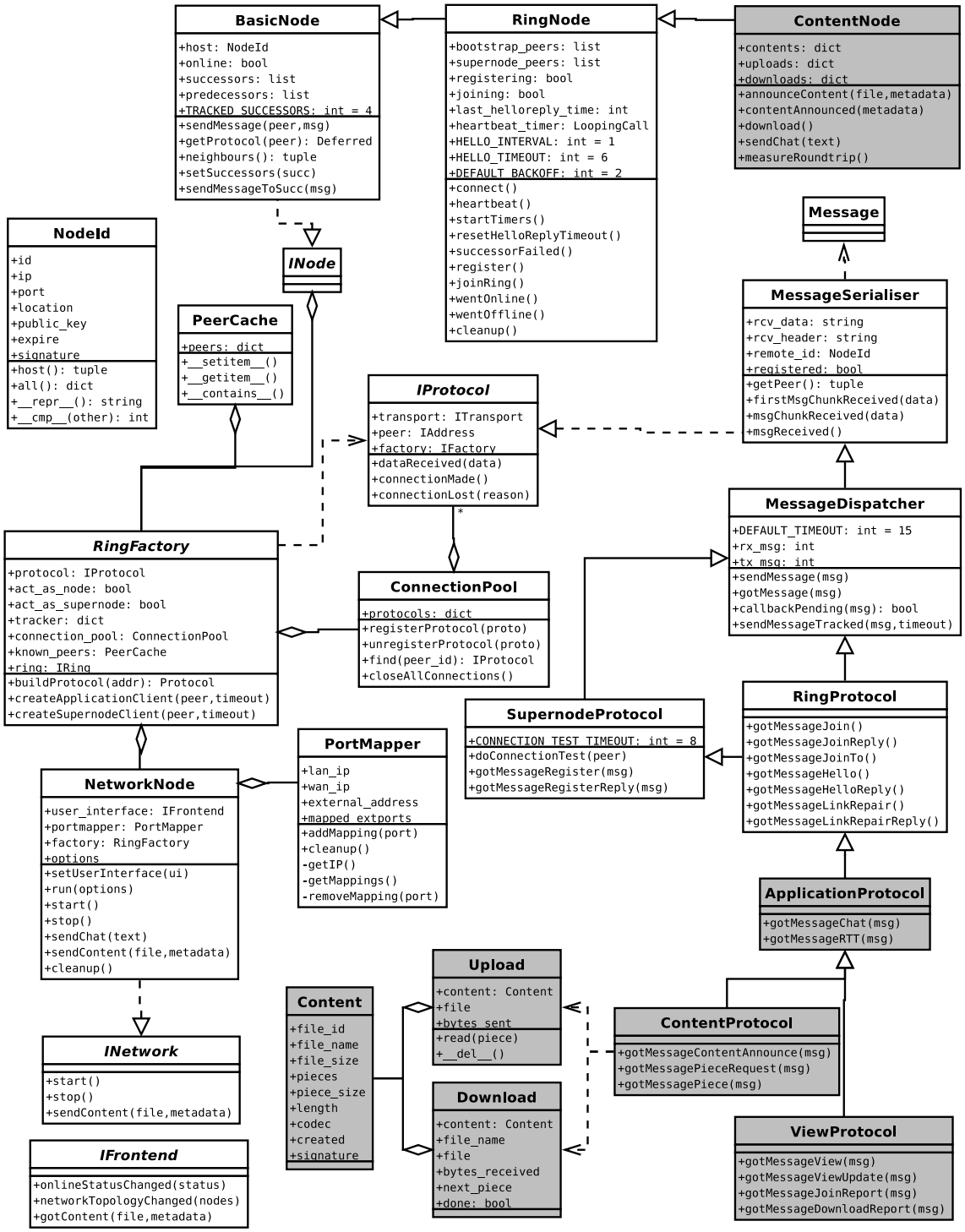

Figure 6.2: Class diagram. White (middleware) and grey (application) classes

Bibliography

[1] Thom Kubli. global contentifier, November 2006. [Online; accessed 23-August-2007] `http://www.khm.de/~kubli/contentifier/`.

[2] *A Survey of Ring-Building Network Protocols Suitable for Command and Control Group Communications*. Conference on Sensors, and Command, Control, Communications, and Intelligence (C3I) Technologies for Homeland Security and Homeland Defense IV, March 2005.

[3] Jun Wang, William Yurcik, Yaling Yang, and Jason Hester. Multiring techniques for scalable battlespace group communications. *IEEE Communications Magazine*, 43(11):124–133, November 2005.

[4] John Risson, Ken Robinson, and Tim Moors. Fault tolerant active rings for structured peer-to-peer overlays. In *LCN '05: Proceedings of the The IEEE Conference on Local Computer Networks 30th Anniversary*, pages 18–25, Washington, DC, USA, 2005. IEEE Computer Society.

[5] Ayman Shaker and Douglas S. Reeves. Self-stabilizing structured ring topology p2p systems. Technical Report 2005-25, June 2005.

[6] Ahmed Sobeih, William Yurcik, and Jennifer C. Hou. Vring: A case for building application-layer multicast rings (rather than trees). In *IEEE Computer Society's 12th Annual International Symposium on Modeling, Analysis, and Simulation of Computer and Telecommunications Systems*, pages 437–446, Washington, DC, USA, October 2004. IEEE Computer Society.

[7] M. Junginger and Yugyung Lee. The multi-ring topology-high-performance group communication in peer-to-peer networks. In *Peer-to-Peer Computing,*

2002. (P2P 2002). Proceedings. Second International Conference on, pages 49–56, 2002.

[8] Atul Singh and Mads Haahr. Creating an adaptive network of hubs using schelling's model. *Commun. ACM*, 49(3):69–73, 2006.

[9] Giscard Wepiwe and Plamen L. Simeonov. A concentric multi-ring overlay for highly reliable p2p networks. In *NCA '05: Proceedings of the Fourth IEEE International Symposium on Network Computing and Applications*, pages 83–90, Washington, DC, USA, July 2005. IEEE Computer Society.

[10] Anwitaman Datta, Sarunas Girdzijauskas, and Karl Aberer. On de bruijn routing in distributed hash tables: There and back again. In *P2P '04: Proceedings of the Fourth International Conference on Peer-to-Peer Computing (P2P'04)*, pages 159–166, Washington, DC, USA, 2004. IEEE Computer Society.

[11] D. A. Agarwal, L. E. Moser, P. M. Melliar-Smith, and R. K. Budhia. The totem multiple-ring ordering and topology maintenance protocol. *ACM Transactions on Computer Systems*, 16(2):93–132, May 1998.

[12] Jianxu Shi and John P. Fonseka. Hierarchical self-healing rings. *IEEE/ACM Trans. Netw.*, 3(6):690–697, 1995.

[13] Anders Weberg. P2p art - the aesthetics of ephemerality, September 2006. [Online; accessed 23-August-2007] `http://p2p-art.com/`.

[14] Scott Draves. electric sheep, 1999. [Online; accessed 23-August-2007] `http://www.electricsheep.org/`.

[15] Marco Mamei. Creating overlay data structures with the tota middleware to support content-based routing in mobile p2p networks. In *HOT-P2P '04: Proceedings of the 2004 International Workshop on Hot Topics in Peer-to-Peer Systems*, pages 74–79, Washington, DC, USA, October 2004. IEEE Computer Society.

[16] Sylvia Ratnasamy, Brad Karp, Li Yin, Fang Yu, Deborah Estrin, Ramesh Govindan, and Scott Shenker. Ght: a geographic hash table for data-centric

storage. In *WSNA '02: Proceedings of the 1st ACM international workshop on Wireless sensor networks and applications*, pages 78–87, New York, NY, USA, 2002. ACM Press.

[17] Dimitri Konstantas and Alfredo Villalba. Hovering information : a new paradigm for sharing delocalized information. Technical report, Department of Information Systems, August 2007.

[18] Giovana Di Marzo, Alfredo Villalba, and Dimitri Konstantas. Dependability requirements for hovering information. Technical report, Department of Information Systems, August 2007.

[19] Owen O'Flaherty. A mobility-aware file system - file availability in a mobile-aware, context aware environment. Master's thesis, University of Dublin, Trinity College, September 2005.

[20] Anne Pascual and Marcus Hauer. Minitasking - a visual gnutella client. In *IV '03: Proceedings of the Seventh International Conference on Information Visualization*, page 115, Washington, DC, USA, 2003. IEEE Computer Society.

[21] Venkata N. Padmanabhan and Lakshminarayanan Subramanian. An investigation of geographic mapping techniques for internet hosts. *SIGCOMM Comput. Commun. Rev.*, 31(4):173–185, 2001.

[22] Peter Gutmann. Pki: It's not dead, just resting. *Computer*, 35(8):41–49, 2002.

[23] Anwitaman Datta, Manfred Hauswirth, and Karl Aberer. Beyond "web of trust": Enabling p2p e-commerce. In *Conference on Electronic Commerce*, pages 303–312, June 2003.

[24] Thomas Wölfl. Public-key-infrastructure based on a peer-to-peer network. In *HICSS '05: Proceedings of the Proceedings of the 38th Annual Hawaii International Conference on System Sciences (HICSS'05) - Track 7*, page 200.1, Washington, DC, USA, 2005. IEEE Computer Society.

[25] Thomas Wölfl and Sven Wünschmann. Public-key-infrastrukturen in einer peer-to-peer-umgebung. In *GI Jahrestagung (2)*, pages 643–647, 2005.

[26] Adam Back. Hashcash - a denial of service counter-measure. Technical report, August 2002.

[27] Adam Back. Hashcash.org, 2007. [Online; accessed 23-August-2007] `http://hashcash.org/`.

[28] Miguel Castro, Peter Druschel, Ayalvadi Ganesh, Antony Rowstron, and Dan S. Wallach. Secure routing for structured peer-to-peer overlay networks. In *OSDI '02: Proceedings of the 5th Symposium on Operating Systems Design and Implementation*, pages 299–314, 2002.

[29] Jerome H. Saltzer, David P. Reed, and David D. Clark. End-to-end arguments in system design. *ACM Transactions on Computer Systems*, 2(4):277–288, November 1984.

[30] Bryan Ford, Dan Kegel, and Pyda Srisuresh. Peer-to-peer communication across network address translators. In *Proceedings of the 2005 USENIX Technical Conference*, 2005.

[31] Divmod vertex, 2007. [Online; accessed 23-August-2007] `http://divmod.org/trac/wiki/DivmodVertex`.

[32] Marling Engle and Javed I. Khan. Vulnerabilities of p2p systems and a critical look at their solutions. Technical report, Internetworking and Media Communications Research Laboratories, 11 2006.

[33] Philipp von Weitershausen and P. J. Eby. *Web Component Development with Zope 3*. Springer-Verlag New York, Inc., Secaucus, NJ, USA, 2007.

[34] Python Software Foundation. Python programming language – official website, 2007. [Online; accessed 23-August-2007] `http://python.org/`.

[35] Glyph Lefkowitz and Itamar Shtull-Trauring. Network programming for the rest of us. In *ATEC'03: Proceedings of the USENIX Annual Technical Conference 2003 on USENIX Annual Technical Conference*, pages 14–14, Berkeley, CA, USA, 2003. USENIX Association.

[36] Twisted matrix labs, 2007. [Online; accessed 23-August-2007] http://twistedmatrix.com/.

[37] D. Crockford. The application/json Media Type for JavaScript Object Notation (JSON). RFC 4627 (Informational), July 2006.

[38] Guido van Rossum and Phillip J. Eby. Coroutines via Enhanced Generators. PEP 342, May 2005.

[39] Raphaël Slinckx. Nattraverso, August 2005. [Online; accessed 23-August-2007] http://raphael.slinckx.net/nattraverso.php.

[40] Jean-Marc Valin. The speex codec manual: Version 1.2 beta 2, May 2007. [Online; accessed 23-August-2007] http://www.speex.org/.