Clippee: A Large-Scale Client/Peer System

Keno Albrecht, Ruedi Arnold, Roger Wattenhofer
{kenoa,rarnold,wattenhofer}@inf.ethz.ch
Department of Computer Science, ETH Zurich, 8092 Zurich, Switzerland

Abstract— **This paper introduces a client/peer architecture providing services similar to Group Communication systems, but with the scalability of peer-to-peer systems and its dynamic behavior. We present Clippee, a first prototype of this client/peer architecture. Clippee employs a data replication scheme based on an optimistic use of locks, without running an expensive consensus protocol. We believe that such a "best effort" approach is an important key to large-scale communication systems. We further show the applicability of this approach through experimental measurements conducted with our prototype.**

I. Introduction

A. Client/Server vs. Peer-to-Peer

The formerly predominant client/server networking paradigm has recently been queried by distributed systems dubbed "peer-to-peer." Peer-to-peer computing takes advantage of existing desktop computing power and networking connectivity, allowing off-the-shelf machines to leverage their collective power beyond the sum of their parts. In a peer-to-peer system, peers (a blend of clients and servers) share computer resources and services by direct exchange. Currently available peer-to-peer systems like Gnutella, Kazaa, or eDonkey primarily focus on the task of sharing multimedia data. With the growing popularity of these systems, the research community has developed schemes how to structure the data in an overlay network; the same basic idea can be found in a variety of proposals (e.g. the system of Plaxton et al. [12], Chord [15], and Pastry [14]). Some of these systems have become full-blown file systems (e.g. Farsite [1]).

Peer-to-peer systems overcome the major drawbacks of client/server systems: (i) Peer-to-peer systems are more dependable, reliable, and available because there is no single point of failure – instead information is usually replicated at several peers. (ii) Peer-to-peer systems scale better because there is no single point of access - there is no extraordinary machine called server, and also no server farm. Almost as a consequence of (i) and (ii), (distributed) denial of service attacks are less of a problem in peer-to-peer than in client/server systems.

The work presented in this paper was supported (in part) by the Hasler Stiftung under grant number 1828 and the Swiss National Science Foundation under grant number 21-66768.01.

However, peer-to-peer systems are not without difficulty themselves. Generally, in a client/server system, the server needs to trust the clients, and vice versa also the clients need to trust the server. In a peer-to-peer system, this one-to-many organization is not available. Instead every peer must trust every other peer, and since in principle any IP-capable machine can become a peer, peers in essence need to blindly trust everything in the Internet.

Moreover, peer-to-peer systems are no egalitarian utopia. Measurements show that many real-existing peers "take" more than they "give". In the long run it is not clear that the concept of "free sharing" can survive in the mostly anonymous Internet cosmos with mostly selfish owners of the peers.

Finally, a peer-to-peer system is not homogeneous: Some peers have much more bandwidth, memory, storage, and/or processing power than others. Some peers are even placed behind firewalls or network address translation systems and cannot be contacted.

B. Client/Peer

For these three reasons we believe that it might be particularly interesting to study the middle ground between the extremes client/server and peer-to-peer. In this paper, we advocate to take the best of the thesis client/server and the anti-thesis peer-to-peer, and synthesize that into what we christen "client/peer."

A client/peer system consists of two types of machines (or agents): clients and peers. The peers form a peer-to-peer system; the peers can be geographically distributed. However, peers are all owned by the same organization. This solves the problems of trust between the peers and of the egalitarian utopia. For an example of a client/peer system see Figure 1.

Clients of a client/peer system behave like a client in a client/server system. A client can connect to an arbitrary peer. A large number of clients can be connected to the same peer. The contacted peer will ensure that the client receives the requested services and that other peers will

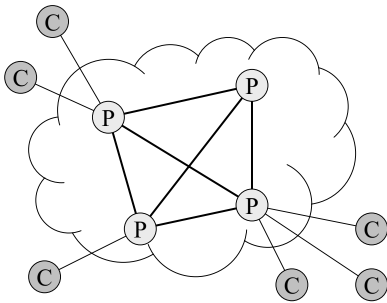


Fig. 1. Illustration of a client/peer system.

be informed about the effects of the client's request. A client can be a simple device behind a firewall, allowing the system to be heterogeneous.

C. Group Communication

Group Communication (GC) is a widely used abstraction for distributed applications. We consider GC an important influence for our work, because of the clear specification of its abstraction model for services and properties. Much research has been done in the GC area over the past almost 20 years. Isis [3] was the first and probably best-known system, followed by Phoenix [11], Horus [16], Ensemble [9] among various others. For a comprehensive study on GC specifications see [6].

GC systems typically offer two main services. The first is group membership: This service provides the notion of views that are sets of active and connected processes, which all processes agree upon. The second service of GC systems is multicast. There are several variants of multicast services with different properties and guarantees. Atomic (reliable and totally ordered) broadcast is typically paid most attention to.

Group Communication systems aim at small numbers of processes which are fully connected. Group Communication is orthogonal to architectures for distributed systems (e.g. client/server or peer-to-peer), as it does not focus on the architecture of systems, but on the services provided. GC has to be seen as a building block for distributed applications or as "middleware."

The abstraction of GC has shown useful in many domains, but it has two well-known drawbacks when applied to large-scale settings:

- **Complete Graph:** GC systems typically work on a set of directly connected processes.
- **Expensive Synchronization:** The number of messages exchanged to reach agreement on a view is in the order of $O(n^2)$, where n is the number of processes in the group [2].

These two reasons motivate our research and distinguish our work from classical GC systems, as our system is based on a peer-to-peer topology and does not concentrate on providing (expensive) group membership and multicast services. In contrast, we focus on scalability and dynamics.

D. Large-Scale Systems

The long-term goal of this project is to tackle the question, which (GC) services can be implemented in a dynamic large-scale system with what effort. There are other approaches to this problem. One is using probabilistic algorithms as in gossip-based protocols. Proposed systems of that kind include SCAMP [8] and Spinglass [4]. Such systems typically employ services like probabilistic atomic broadcast [7] or adaptive gossip-based broadcast [13] and offer some kind of optimistic or "best effort" services, such as the lightweight membership service described in [8]. This is a clear distinction from our approach, since we do not employ gossiping. Neither do we provide an explicit notion of (partial or probabilistic) membership information. Another promising approach is providing multicast services on top of an existing peer-to-peer system, as for example demonstrated by SCRIBE [5].

The goal of our system is to provide services to a large number of clients, where frequent changes within the sets of clients and peers are supported:

- **Scalability:** Our system is aimed to scale better than classical GC systems.
- **Dynamics:** Our system adapts more gracefully to dynamic changes in the topology, such as processes joining and leaving. We have no explicit notion of views[1] nor does our system provide an (expensive) atomic broadcast primitive. However we do provide different services at different costs.

In this paper we introduce Clippee (spoof of "client/peer"), a first prototype system following the client/peer approach. With the current Java implementation, we ran a series of successful experiments.

The remainder of this paper is structured as follows. In Section 2 we present an outline of Clippee. Section 3 summarizes the results of performance measurements, showing the practicability of Clippee. Finally Section 4 concludes the paper.

II. Architecture

In Figure 2 we present our peer-architecture that consists of three basic components: (a) The *network*

[1]There is therefore no need to conduct expensive view-change operations.

Storage

Eager Global Write
Local Read
Lazy Replication Process

Topology

Complete Graph
2-Hop-Routing

Network

Send / Receive Messages
(Re-)Connect

Fig. 2. The peer-architecture consists of a *network*, *topology*, and *storage component*.

component provides connectivity between hosts and primitives for sending and receiving messages, (b) the *topology component* defines an overlay network, and (c) the *storage component* manages data stored in the system. Notice that, although we arrange these services as layers, there is no common interface between them. The reason is that the components depend on each other.[2] Therefore implementations cannot be swapped without reconsidering dependencies. Furthermore layers are allowed to use any other layer's functionality directly, for instance the application can send messages using network operations and bypass the replication or topology layer. We emphasize that the following subsections describe our prototype implementation and thus ongoing work.

A. Model and Assumptions

We consider an asynchronous system with processes that communicate by message passing and fail only by crashing. A message can take an arbitrary long time to arrive at its destination; the processing time of messages is also unbounded.

All peers are connected as a complete graph (except two-hop-routes, see section II-C) acting as one logical server for all clients. Peers can seamlessly join and leave without affecting the system, but at least one peer storing all replicated data is expected to be alive at any time. We guarantee message delivery with at most $n-2$ simultaneous link failures, thus there exists at least a two-hop-route between any pair of peers.

A peer can store small data objects. We provide concurrent read and exclusive write/create operations. Data is fully replicated; we use a quorum-ROWA-A-like mechanism [10] with global locking as a basis for our optimistic replication scheme. Since we are aware of possible inconsistencies, we employ a background process called *lazy replication process* to ensure eventual consistency.

Clients can register with any known (login-)peer. Several clients can login to the same peer simultaneously, thus the number of clients can be much larger than the number of peers. If a login-peer fails, its clients automatically connect to another randomly chosen peer. If no peer is known, user-action is required.[3] Clients can join at any time and leave after an arbitrary period of time without affecting any peer or client. We assume that in absence of any failure a client does not change its login-peer, thus it remains with its once chosen login-peer until (intentionally or unintentionally) disconnecting from it.

A client does not store any data itself. Instead, it can read and write data via its login-peer. A client is allowed to modify an arbitrary number of different objects concurrently.

B. The Network Component

The lowest layer of our architecture handles the delivery of messages between peers over TCP-connections. Each connection uses a single sender thread taking messages from a buffer that stores outgoing messages for this connection. A message is either sent directly to the receiver or – if the direct link failed before – via a two-hop-route using information delivered by the topology component. Incoming messages are handled concurrently by a bounded but adjustable number of threads. Since multithreading in Java does not guarantee fairness between concurrent processes, we do not provide FIFO on a single connection, unless we set the number of threads handling incoming messages to one (which results in a decrease of efficiency). Furthermore, our current prototype drops incoming messages if there is no free thread to handle it. We do not want to use queues to buffer messages, since this leads to long response times (especially under high load), which we want to minimize.

Requests asking for reply messages are handled by employing a timeout mechanism. Missing requests must be handled by its sender, for instance by resending or error handling. Timeout expiration might occur due to crash, dropping of a message, or slow processing. Replies arriving after their timeout has expired are dropped.

[2] For example, an instance of the replication service might rely on FIFO communication over links, whereas not every network component has to support this type of message delivery.

[3] This problem arises in virtually every pure peer-to-peer system and is known as the peer-to-peer bootstrapping problem.

An additional feature is that indirect connections using two-hop-routing try to reestablish direct connections.

C. The Topology Component

Our topology component ensures that every peer is connected to every other peer in the system. All peers are connected as a complete graph. Information about joining peers is sent to every other peer, the joining peer receives a message containing a list of all current peers, and leaving peers (gracefully or by crashing) are detected on every remaining peer by a failure detector. Additionally the information about live peers is spread regularly. Depending on the ID[4], only one of two peers trying to (re)establish a connection is allowed to do so in order to avoid mutual connection setup.

Upon link failures, a two-hop-route is sought for by *pinging* the destination peer via a forward request to all other peers. The first reply (including the new route) is subsequently used until reestablishment of a direct connection by the network layer. Since both peers related to the link failure try to reconnect concurrently, different two-hop-routes might be found. We can guarantee message delivery up to $n-2$ simultaneous link failures, since there exist $n-1$ routes (one direct and $n-2$ two-hop-routes) from every peer to any other peer.

Our system was implemented with considering *virtual failures*; thus we can send messages to a peer to simulate a crash or a link failure.

D. The Storage Component

In our peer-system we can store *small* (in the order of some kBytes) data objects. Data is fully replicated, every peer is storing every data object. Using an eager replication algorithm, the size of data objects linearly increases the time necessary to store it.

A joining peer automatically downloads all data from an already existing peer (its *bootstrapper*). Nevertheless, the joining peer is immediately fully operational: It handles requests to known data objects on its own, whereas requests to unknown objects are "forwarded" to its bootstrapper.

The algorithm we use to manage data is related to the ROWA-A-approach described in [10]: Data objects are always written on all (available) peers. In contrast, the read-operation is a local lookup only.

To write an object, a peer has to acquire a *lock* (token) for that object by sending an *AcquireLock*-message to every peer. The lock is acquired only if all peers grant the lock or do not answer in time. This optimistic approach does neither guarantee unique locks nor concurrent, mutual write-access to the same object. Note that these cases arise very rarely – only upon timeouts or when separated subsystems rejoin.

For safety, we additionally employ a *lazy replication process* that handles and fixes possible inconsistencies in the background by cross-checking (comparing) data objects of randomly chosen peers. This mechanism is also used on failures due to occurrence of inconsistency upon write access. More precisely, a write operation fails only in two cases: A peer denies writing an object, because (i) the writing peer does not own the lock, or (ii) the writing peer wants to write an object that does not have a newer version than the current one. In both cases the lazy replication mechanism is used to fix the inconsistency.

After the lock has been acquired, a peer can update the object until releasing the lock again.

III. Results

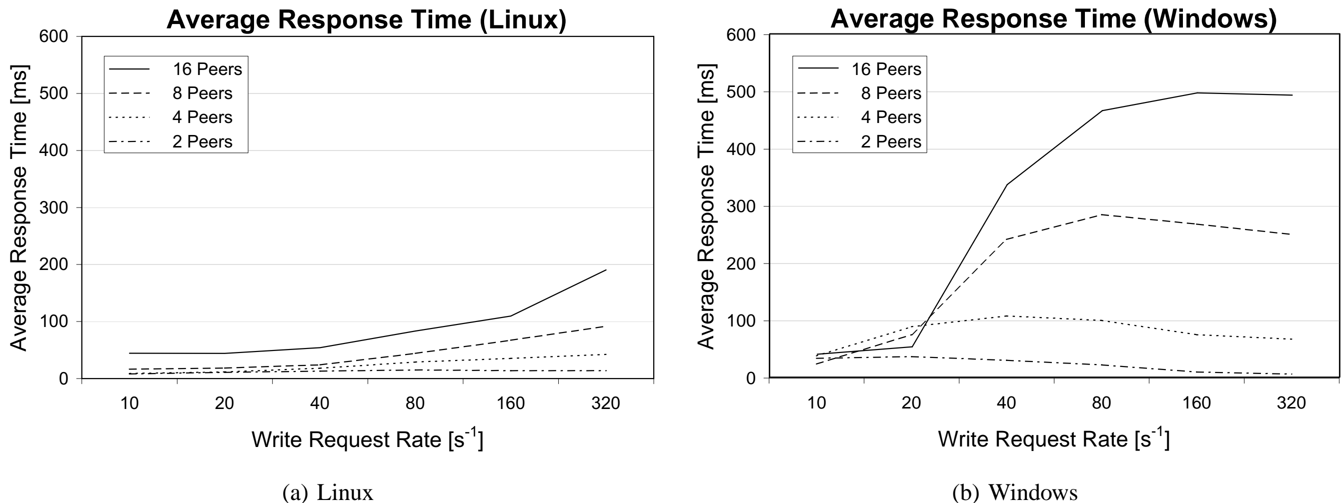
In this section we provide some results of our measurements. Our testing environment consists of PCs having one or two 600 MHz processors, 256 MB of RAM, a 100 MBit/s network adapter, and both Windows 2000 and Red Hat Linux 7.3 installed. The system is implemented in Java using Sun's current SDK 1.4.2. We ran experiments with 2 to 16 peers, one per PC. Clients were evenly spread among all peers, ensuring that the request load was uniformly distributed among the peers. Additionally, we used one machine to act as a *central controller* to coordinate our measurements: All peers and clients were connected to this dedicated machine and received information about the upcoming experiment. We emphasize that the central controller is not necessary for running the system, but simplifies the management of measurements.

We ran various sets of experiments. Due to lack of space we focus on one setup: In this experiment we have a fixed number of clients running on different machines. Each client is assigned a fixed login-peer and sends write requests for random data objects to its login-peer at a given rate, using a Poisson distribution.[5] The total load of requests to the whole system ranges from 10 to 320 write requests per second.

For Linux, Figure 3(a) shows the average response time of write requests as a function of the request rate

[4] We currently use the IP address and the port number of a peer as its unique identifier.

[5] A Poisson distribution is well-suited to model requests from a large client population, since general (non-Poisson) request distributions sum up to almost a Poisson process if a large number of clients acts independently.



(a) Linux

(b) Windows

Fig. 3. The average response time of write requests as a function of the request rate and different numbers of peers.

and different numbers of peers. For small request rates, the average response time is low. With growing request rates, the response time increases proportionally with the number of peers. This is the expected behavior, since with an increasing rate also the number of messages increases and thus the total load of the peers. If peers are overloaded and get more requests than their receiver threads can handle, they start dropping requests. In our experiments, the peers' drop rates (resulting in timeouts on client side) were always less than 1% of all requests.

Although this also holds when using Windows, unexpectedly, Figure 3(b) shows that for high request rates, the curves do not only flatten, but the values even decrease. The average response time at 320 requests per second is for all peer numbers reproducibly better than at 160 requests per second. We currently do not understand this behavior, but we think that it might be due to the different threading and networking mechanisms used by the two operating systems.

As opposed to write requests, where each request is handled by all peers, read requests are handled locally by a peer. For read requests, more peers therefore not only help in reliability, but also in scalability. In fact, read requests are so efficient that we were not able to produce a big enough request load to bring our system to the fringe of operability. A simple back-of-the-envelope computation helps to understand the issue: While a read request triggers 1 message, a write triggers $5(n-1)$. For every write request in our experiment, the Clippee system should therefore be able to handle about $5n$ reads. A system with n peers being able to handle K writes per second should also be able to handle $K(W+5nR)$ read and write requests per second. Here R and $W=1-R$ denote the percentage of reads and writes, respectively. Therefore this system can also cope with $5nK$ reads per second if the system is loaded exclusively with read requests.

IV. Conclusions

In this paper we introduced the client/peer architecture. We presented a first prototype Clippee, a distributed system to provide services in a scalable and dynamic distributed environment. We have shown our system to be operable with no atomic broadcast or consensus protocol, instead employing a simple but effective locking mechanism combined with a (background) update thread to provide eventual consistency.

Our measurements with the current Java implementation have shown that read requests are processed rather fast, since they can be handled locally and their response time does not depend on the number of peers in the system. Write response times on the other hand grow with increasing number of peers.

References

[1] A. Adya, W. J. Bolosky, M. Castro, G. Cermak, R. Chaikena, J. R. Douceur, J. Howell, J. R. Lorch, M. Theimer, and R. Wattenhofer. Farsite: Federated, available, and reliable storage for an incompletely trusted environment. In *OSDI*, 2002.

[2] G. Berket, L. Moser, and P. Melliar-Smith. The intergroup protocols: Scalable group communication for the internet. In *Proceedings of the IEEE Global Telecommunications Conference GLOBECOM, Global Internet '98 Mini-Conference Record*, pages 100–105, Sydney, Australia, November 1998.

[3] K. P. Birman. Isis: A system for fault-tolerant distributed computing. Technical report, Cornell University, Department of Computer Science, April 1986.

[4] K. P. Birman, R. van Renesse, and W. Vogels. Spinglass: Secure and scalable communications tools for mission-critical computing. In *International Survivability Conference and Exposition. DARPA DISCEX*, Anaheim, California, June 2001.

[5] M. Castro, P. Druschel, A. Kermarrec, and A. Rowstron. SCRIBE: A large-scale and decentralized application-level multicast infrastructure. *IEEE Journal on Selected Areas in communications (JSAC)*, 2002.

[6] G. V. Chockler, I. Keidar, and R. Vitenberg. Group communication specifications: A comprehensive study. In *ACM Computing Surveys*, number 33(4), pages 1–43, December 2001.

[7] P. Felber and F. Pedone. Probabilistic atomic broadcast. In *21th IEEE Symposium on Reliable Distributed Systems (SRDS)*, 2002.

[8] A. J. Ganesh, A.-M. Kermarrec, and L. Massoulie. SCAMP: Peer-to-peer lightweight membership service for large-scale group communication. In *Proceedings of the 3rd International workshop on Networked Group Communication*, London, UK, November 6-9 2001.

[9] M. Hayden and R. van Renesse. Optimizing layered communication protocols. Technical report, Dept. of Computer Science, Cornell University, November 1996.

[10] A. Helal, A. Heddaya, and B. Bhargava. *Replication Techniques in Distributed Systems*. Kluwer Academic Publishers, 1986.

[11] C. P. Malloth, P. Felber, A. Schiper, and U. Wilhelm. Phoenix: A toolkit for building fault-tolerant, distributed applications in large scale. In *Workshop on Parallel and Distributed Platforms in Industrial Products*, October 1995.

[12] C. G. Plaxton, R. Rajaraman, and A. W. Richa. Accessing nearby copies of replicated objects in a distributed environment. In *ACM Symposium on Parallel Algorithms and Architectures*, pages 311–320, 1997.

[13] L. Rodrigues, S. Handurukande, J. Pereira, R. Guerraoui, and A.-M. Kermarrec. Adaptive gossip-based broadcast. In *Proceedings of the International Conference on Dependable Systems and Networks (DSN)*, page (accepted for publication), San Francisco (CA), June 2003.

[14] A. Rowstron and P. Druschel. Pastry: Scalable, decentralized object location, and routing for large-scale peer-to-peer systems. *Lecture Notes in Computer Science*, 2218, 2001.

[15] I. Stoica, R. Morris, D. Karger, F. Kaashoek, and H. Balakrishnan. Chord: A scalable Peer-To-Peer lookup service for internet applications. In *SIGCOMM*, 2001.

[16] R. van Renesse, K. P. Birman, and S. Maffeis. Horus: A flexible group communication system. In *Commun. ACM*, number 39, pages 76–83, April 1996.