The Design of a Robust Peer-to-Peer System

Rodrigo Rodrigues, Barbara Liskov, Liuba Shrira*
MIT Laboratory for Computer Science
{rodrigo, liskov, liuba}@lcs.mit.edu

Abstract

Peer-to-peer (P2P) overlay networks have recently become one of the hottest topics in OS research. These networks bring with them the promise of harnessing idle storage and network resources from client machines that voluntarily join the system; self-configuration and automatic load balancing; censorship resistance; and extremely good scalability due to the use of symmetric algorithms. However, the use of unreliable client machines leads to two defects of these systems that precludes their use in a number of applications: storage is inherently unreliable, and lookup algorithms have long latencies. In this paper we propose a design of a robust peer-to-peer storage service, composed not of client nodes, but server nodes that are dedicated to running the peer-to-peer application. We argue that our system overcomes the defects of peer-to-peer systems while retaining their nice properties with the exception of utilizing spare resources of client machines. Our system is capable of surviving arbitrary failures of its nodes (Byzantine faults) and we expect it to perform and scale well, even in a wide-area network.

1 Introduction

We have witnessed the recent emergence of a number of peer-to-peer (P2P) distributed systems, i.e., systems in which all nodes have identical responsibilities and all communication is symmetric. Successful applications of these systems include content sharing [7] and large-scale storage systems [6].

Peer-to-peer computing offers several advantages over other traditional distributed systems, such as automatic load balancing and self-organization. But perhaps the most valuable feature of these systems is that, due to the symmetric nature of peer-to-peer, the desirable properties of the system can scale when new nodes are added to the system. These properties include (but are not limited to) performance, availability, and fault-tolerance.

Part of the success of these systems comes from their ability to harness idle storage and network resources, offered by everyone who is willing to participate in the system. Unfortunately, such resources are inherently unreliable: we cannot control what code is running in these machines, and they will join and leave the network frequently. This leads to two essential defects of P2P storage systems: they offer weak guarantees in terms of storage reliability; and they must include complicated lookup algorithms (with very high latency) to allow low cost routing information updates in the presence of frequent joins and leaves.

In this paper we describe the design of a P2P storage system that solves these problems. Our system provides reliable storage in the presence of failstop failures. In addition it also performs reliably in the presence of Byzantine failures: the storage service will continue to work correctly even if faulty components behave arbitrarily. Our system overcomes the main problems of traditional P2P storage systems while maintaining their essential design philosophy and the advantages that derive from it.

Our system has a hybrid architecture, consisting of a set of servers — not unreliable client machines that participate in the system intermittently — acting as the P2P nodes, and a *configuration service* (CS). The CS runs on a set of special servers (presumably less failure-prone than P2P nodes). It is responsible for computing the current configuration (including removing faulty P2P nodes from the system), and informing the P2P nodes about the current configuration.

Not all applications require the kind of robustness this system provides. In particular, applications like the Cooperative File System (CFS) [3] assume that the file system is created in a secure place and the untrusted peer-to-peer storage is used only to publish information. Therefore, if the peer-to-peer system loses state, the information can be refreshed from the original source. Our techniques are required, however, for applications that store their actual state, rather than a copy of the state, in the P2P system. Examples

* Department of Computer Science, Brandeis University

This research was supported by DARPA under contract F30602-98-1-0237 monitored by the Air Force Research Laboratory. Rodrigo Rodrigues is supported by a Praxis XXI fellowship.

of such applications include various kinds of archival services, ranging from digital libraries to archives for file systems and object-oriented systems; mail services; key distribution services; and databases that store large amounts of information such as astronomical data.

The next section discusses P2P computing and motivates our approach by explaining why P2P systems as presently designed cannot tolerate Byzantine-faulty nodes and what is needed to fix the problem. Section 3 presents our system model and lists our assumptions. The configuration service is presented in Section 4 and the P2P nodes in Section 5. We conclude in Section 6.

2 Motivation

This section presents a brief description of P2P systems and then explains why these systems cannot tolerate Byzantine failures. Then it describes what is necessary to solve the problems: P2P servers that do proactive recovery and a configuration service that determines the current configuration using Byzantine agreement protocols and propagates the information to the P2P nodes.

2.1 Peer-to-Peer Computing

Recently proposed P2P storage systems provide the abstraction of a distributed hash table [12, 14, 18, 19]. Each data item is assigned an ID, and the system provides the ability to store the item with that ID and to retrieve it later. Applications choose the IDs and this is typically done in a way that allows the data to be self-verifying.

The system is composed of nodes each of which is assigned an ID. The system stores an item at a node or nodes based on a computation that takes into account the item ID and the node IDs.

The functionality of the P2P nodes is divided into a lookup layer that locates the nodes responsible for an item given its ID, and a storage layer built on top of the lookup layer that provides the distributed hash table abstraction. Each of these layers provides both client and server functionalities: the client storage layer uses the client lookup layer to locate servers that hold the desired data. The server storage layer is responsible for storing the data items, maintaining proper levels of replication, and performing caching. The server storage layer and lookup layer interact in order to maintain the correct mapping between the servers that are present in the system and the data they hold.

Node IDs are typically computed by applying a collision-resistant base hash function such as SHA-1 [16] to information about the node, such as its IP address, resulting in an m-bit identifier. Data item IDs are also of the same length (and are often computed by hashing the content of the item).

Different peer-to-peer lookup systems use different criteria to determine which node is responsible for storing a particular data item. For instance, Chord [18] is based on consistent hashing [8], where identifiers are ordered in an identifier circle modulo 2^m, and the data item with ID i is assigned to the first node whose identifier is equal to or follows i in the identifier space (called the *successor node* of ID i). Pastry [14], on the other hand, allows applications to store data items in the nodes with IDs numerically closest to the ID of item being stored. Without loss of generality, we will assume the use of consistent hashing in our system description. Changing the system to use other mappings should be straightforward.

The storage layer builds a reliable storage service on top of the lookup layer. Reliability cannot be achieved without replication, and therefore this layer must make decisions on where to store the replicas of the data. Typically, there is a great deal of interdependence between the replica placement decisions and the details of the lookup algorithms. For instance, the DHash layer built on top of Chord [3] stores the data item not only at the successor of its ID i but also at the next n successors. These are easy to determine since the Chord layer maintains a list of successors for each node. Similarly, PAST [15] stores replicas on the n nodes whose ID are numerically closest to the item ID i, which is also information contained in the data structures maintained by the Pastry lookup algorithm.

Our system can easily adapt to any placement strategy. We will assume, without loss of generality, that we store each data item at the n successors of ID i in the ID space (similarly to DHash). We believe that changing this placement strategy does not affect the properties of our system, as long as the strategy is based on randomly-chosen node IDs. This is important so that the likelihood of replicas of each data item failing is as independent as possible. Random node IDs avoids, for instance, having all replicas in the same LAN, which would make a simple disconnection of the LAN from the Internet sufficient to make the data unavailable.

2.2 Problems in Peer-to-Peer Systems

Most currently proposed peer-to-peer systems are designed to tolerate failstop failures (e.g., [14, 18]). They assume replicas fail by stopping or omitting some steps. Two aspects of these algorithms make them particularly vulnerable to malicious attacks.

First, there is no admission control. This implies that an attacker can join the system with multiple personalities that offer a large amount of resources (even if the attacker does not own such resources, as described in [5]) and this way the attacker can control a substantial fraction of the nodes and increase the probability of a successful attack.

Second, all nodes trust the configuration information they hear from other nodes to be correct. If this information is incorrect (because it comes from a Byzantine-faulty node), this can cause a client request to be diverted to a parallel, internally consistent system formed only of malicious nodes that pretend to store the data correctly but can choose to not allow correct retrieval [17]. Or even more seriously, a P2P node might update its routing state incorrectly, to reflect a bogus configuration change reported by a malicious node.

Solving this second problem is difficult, and its solution motivates the main architectural choices of our system: we need a way to reliably detect faulty nodes and to remove them from the system. As we will explain, the configuration service provides this ability.

2.2.1 Detecting Failed Nodes

Knowing a node is faulty is difficult even if we are only trying to detect failstop failures. Typically, fault detection can be done by some sort of ping protocol where you periodically test that the node is alive and reachable. Assuming the node is faulty when you cannot ping it is not reasonable in the presence of denial of service attacks. Still, if we assume denial of service of attacks are bounded we can reason this way.

Detecting Byzantine faults that result, for instance, from a malicious attack is much harder. This is so because nodes can appear to behave properly even if they have been compromised, and therefore obtaining a proper reply from a node does *not* imply it is correct. Also, it is impossible to check if a node is correct by inspecting its state, since a faulty node could appear correct when inspected and misbehave when replying to clients. Thus, the only way to detect Byzantine faults by probing is if the probes are indistinguishable from client requests. But even with such probes, we can't guarantee to detect Byzantine-faulty nodes in time, because an attacker can compromise more and more nodes, making them behave properly until enough have been compromised to cause the system to malfunction. In other words, it is impossible to get rid of Byzantine faulty nodes in a timely way using probes.

Thus instead of probing, we rely on a proactive recovery mechanism [2] to make Byzantine-faulty nodes behave correctly again. In this scheme, all nodes in the system are recovered proactively and frequently, even if there is no reason to suspect they are faulty. When a node is recovered, it is rebooted and restarted from a read-only disk that contains a correct version of the code. Then it is brought up-to-date by fetching its missing or incorrect parts of the service state from nodes that contain copies of that state. This mechanism allows us to assume that nodes are Byzantine faulty only for short periods, so that we do not need to worry about detecting and removing Byzantine faulty nodes. Thus we will concentrate on removing unreachable nodes from the system.

Note that proactive recovery makes it unlikely that P2P nodes could be client machines that voluntarily join the system and provide their resources to it. Instead they must be server machines dedicated to handling the distributed application. However, as discussed earlier, there are other reasons why we want to use servers as the P2P nodes. Server machines are less failure-prone than client machines (which minimizes the probability of enough machines being compromised so that the system becomes unavailable), and they are not constantly joining and leaving the system, which facilitates management of the routing state. However, there can still be huge numbers of nodes in the system, provided by many different organizations; they can still run symmetric protocols; and the nodes can still be distributed across a wide area, allowing them to survive catastrophic failures (from earthquakes to terrorist attacks).

2.2.2 Removing Faulty Nodes

We still need to figure out how to decide what nodes should be removed from the system and we also need to propagate this information to the P2P nodes in a way that cannot be subverted by Byzantine-faulty nodes.

Deciding which nodes are faulty requires some form of agreement, since we cannot trust an individual node to be correct and make the right decision. Therefore the decision must be made by a group of replicas that carry out an agreement protocol that is robust in the presence of Byzantine failures. These replicas must run a Byzantine agreement protocol [1, 2] to agree on the correct state of the configuration.

Our architecture uses the configuration service (CS) for this purpose. The CS controls membership in the current P2P configuration and periodically notifies the P2P nodes of configuration changes. It uses an agreement protocol to decide on changes, and it propagates configuration information to the P2P nodes, authenticated using digital signatures so that the P2P nodes can be certain that the information is correct.

The CS could run on a subset of the P2P nodes. It could only run on a subset, rather than on all the P2P nodes, because the CS replicas need to communicate with all other nodes, and they need to carry out agreement protocols, so this would be impractical if all P2P nodes were involved. The subset might be selected statically, but that would go against our self-configuring design principle. Or we could imagine that it is selected dynamically: part of defining the next configuration is choosing the subset of nodes that will be the CS for that configuration.

However there are advantages to keeping the CS separate

from the P2P nodes. The CS nodes can be more reliable than the P2P nodes, with hardened security (e.g., physically isolated, geographically diverse, running different software and with different administrators). In addition the CS nodes have lots of work of their own to do; having P2P nodes do this work might lead to excessive load.

3 System Model

We assume an asynchronous distributed system where nodes are connected by a network. The network may fail to deliver messages, delay them, duplicate them, or deliver them out of order. We use a Byzantine failure model, i.e., faulty nodes may behave arbitrarily.

To authenticate communication in the presence of Byzantine faults, we rely on cryptographic techniques that an adversary cannot subvert. Not only do we need such protocols for communication within the CS and from the CS to the P2P nodes, we also require them occasionally for communication between P2P nodes (as discussed below). Therefore we assume that each node (both CS and P2P) has a secure cryptographic co-processor (which prevents exposure of a node's private key), a read-only disk where it stores the correct service code, and a watchdog timer that triggers recoveries. These are common assumptions for Byzantine fault tolerance algorithms [2].

We assume that all nodes in the system initially get to know the identity and public keys of the replicas in the CS using an out-of-band mechanism. When admitting a P2P node in the system, the CS gets to know its public key as well. This information is propagated to the P2P nodes as part of the configuration information.

We allow for a very strong adversary that can coordinate faulty nodes, delay communication, or delay correct nodes in order to cause the most damage to the replicated service. We do assume that the adversary cannot delay correct nodes indefinitely, and cannot cause an arbitrary delay to messages that are sent to reachable nodes (i.e., there are bounds on the duration of a denial-of-service attack).

4 The Configuration Service

The CS is responsible for determining the current configuration of the system, and propagating this information to the P2P nodes, so that they know what other nodes to contact to store or retrieve data. The CS replicas carry out a Byzantine-fault-tolerant protocol [1] to ensure that they agree about configuration state; this is necessary to ensure that the configuration state is correct and consistent.

This service performs four main functions that are described in the next sections.

4.1 Admission Control

The CS controls nodes entering the system, since otherwise, as discussed earlier, a malicious party can subvert the system.

The simplest way to do admission control is for the CS to maintain a list of authorities who are permitted to add nodes to the system. Each request to add a node must be signed by one of these authorities.

In addition, the CS provides a way to add and remove authorities.

When a node joins the system, the CS must be informed about its ID, its IP address, and its public key. This information will be propagated to the P2P nodes in the next configuration description.

4.2 Node Monitoring

The CS monitors the availability and reachability of the nodes using a ping protocol. Each CS replica must do its own monitoring of all the P2P nodes. This is needed so that it can form its own view of which nodes are faulty; then it will be able to decide whether a configuration change proposed by some other replica is reasonable.

CS replicas can do monitoring by sending pings to the P2P nodes. Alternatively, pings could be initiated by the P2P nodes. A good time to do this is when the P2P node restarts after proactive recovery since at that moment, the node is up and not faulty. We plan to use a combination of these techniques since each has its advantages. If pings are initiated by the P2P nodes there is less traffic since pings need not be acknowledged. Initiating the pings by the CS allows us to control the periodicity of the pings. This way we can increase the frequency of pings when we suspect a node is down, so that we can verify this more credibly.

The results of the pings are inserted in a *liveness database* local to the CS replica.

4.3 Deciding on a New Configuration

Periodically the CS nodes must decide on a new configuration. The new configuration will contain all the new nodes that joined the system since the last configuration was produced. The new configuration will *not* contain nodes that the CS replicas agree are faulty.

CS nodes scan the liveness database and try to evict potentially faulty nodes from the system. Having the CS nodes agree on evicting someone is not trivial, as different nodes will have different values for how long each node has been unreachable. We solve this using the non-deterministic choices validation mechanism proposed in [13]. CS nodes initiate a node eviction operation on the CS group if they haven't heard from a node for longer than $T + \epsilon$ time units.

The decision to evict a node is a non-deterministic choice, and therefore it needs to be validated by other CS replicas. They should accept the operation if they haven't heard from the same node for longer than T time units. This approach ensures that most eviction operations will succeed.

The eviction threshold $T + \epsilon$ is chosen to be longer than the assumed bound on denial-of-service attacks. It should be enough longer that the probability of evicting a non-faulty node because the attack prevents communication with it is small.

4.4 Propagating Configuration Information

The CS produces new configurations periodically (e.g., once every hour). Doing this only once in a while makes the system much more practical. We require that new configurations be generated often enough to preserve the following correctness condition for the P2P nodes:

At any moment, for any group of $n = 3f + 1$ replicas of a data item, that group contains no more than f faulty replicas.

A configuration description includes start and expiration times. The new configuration has a start time equal to the expiration time of the previous configuration. We assume that all participants have loosely synchronized clocks that allow them to perceive similar configuration intervals. The assumption about loosely synchronized clocks is a reasonable one for current systems due to the use of clock synchronization protocols [11]. (If these protocols do not provide adequate level of fault-tolerance, GPS can be used instead.)

New configurations must be signed by the CS and propagated to all the nodes, e.g., using gossip methods [4] to avoid overloading the CS. Signing configurations is not trivial, since an attacker can compromise one replica of the CS at a time, and have each replica sign wrong future configurations that are later combined to form a valid signature. We could solve this by having all P2P nodes read the configuration state from the CS nodes (from $2f + 1$ of them) periodically, but this is a heavy weight solution that precludes the use of gossip methods and increases load on the CS. One possible solution is to employ threshold cryptography methods for signing certificates as proposed in [20].

We have choices about what configuration information to propagate to which nodes. For example, we could partition the configuration information and each P2P node would learn only the information needed for it to carry out its base algorithm. Thus in a Chord system [18], nodes would be given their successors and their fingers.

But having to deal with incomplete configuration information is a problem in P2P systems. Traditional peer-to-peer systems have to cope with nodes frequently joining and leaving the network, and therefore try to limit the amount of routing state that has to be updated when configuration changes occur. The penalty for this is having to use lookup algorithms with high latency: each lookup involves contacting a substantial number — typically $O(log(N))$ — of nodes [12, 14, 18, 19]. In our system, however, we assume that nodes join and leave the system much less frequently than client machines in traditional P2P systems, and these node addition and removal operations are grouped together in even less frequent configuration changes dictated by the CS.

Therefore it seems reasonable to take advantage of the CS to simplify the routing algorithm by disseminating the entire configuration information to all the P2P nodes. Thus, routing decisions can be made locally, avoiding the latency of contacting a series of nodes.

The main cost of this approach is that the P2P nodes must store all this information. But actually the amount of storage required is small, even when we scale to hundreds of thousands of nodes. If we assume that the configuration consists, for each node in the system, of a 160 bit node identifier (based on a SHA-1 cryptographic hash function), plus its IP address, port number, and 1024 bit RSA public key, then the entire configuration will fit in approximately 14.7 megabytes, when we scale to $100,000$ nodes. This information can easily fit in main memory.

Transmitting the information from the CS to the P2P nodes is not an issue since this can be done using diffs. The configuration description must contain a signed certificate containing the start and end time of the configuration and a fingerprint of its current state, so that nodes can communicate about configurations reliably without having to send the entire configuration.

In addition, the CS needs to allow P2P nodes to read the entire configuration; this is necessary when a node first starts using the system, or if it becomes very out of date. This is a large amount of information to be transmitted, but some of it is highly compressible (e.g., IP adresses and port numbers). Also, we could devise a mechanism where only the current configuration and specific public keys are obtained immediately, and the remaining public keys are obtained in the background. After a node is temporarily disconnected we can minimize the amount of information transmitted by using Merkle trees [10] to determine exactly the subset of the configuration that is out-of-date.

5 Peer-to-Peer Nodes

This section describes the processing at the P2P nodes. We designed our system with an application like CFS in mind, but extended the model to provide robustness. In the future, we intend to investigate what aspects of this design are not suitable for other applications and refine it to overcome these problems.

Figure 1 illustrates the software structure of the P2P

nodes. The figure makes a distinction between clients and servers. However, a server machine can also act as a client, similar to what happens in other P2P systems; in this case the application layer must be prepared for the unavailability of the node during proactive recovery. Alternatively, there may be dedicated clients that do not implement server functionality.

Other than the possibility of dedicated clients, the software structure is similar to what is used in previous P2P systems. In the description of the P2P nodes, we will use the term "client" to refer to the client functionality of the P2P nodes, which may correspond either to the client subset of the P2P node functionality, or to a dedicated client.

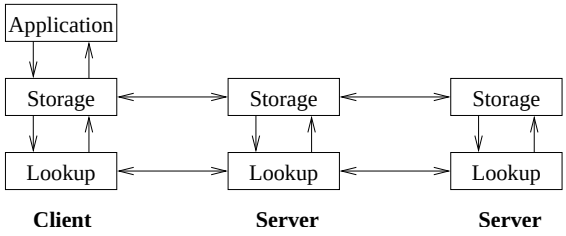

Figure 1. Software Structure

5.1 Lookup Layer

Periodically, the lookup layer receives information from the CS about the latest configuration. The lookup layer will notify the storage layer of this fact. In the new configuration, it is likely that the set of P2P nodes storing certain data items will change; the storage layer must do state transfer (described in Section 5.2.2) to move those items to the nodes that now store them.

The lookup layer on the client side also maintains information about the latest configuration. This enables clients to go directly to the P2P nodes that hold the data of interest: the storage layer at the client side asks the lookup layer for the replicas that hold the data item with a particular ID, and then it can contact the storage layer at the replicas directly.

P2P nodes obtain the configuration information from the CS the first time they join the system, but after this they can obtain new configurations directly from other nodes.

5.2 Storage Layer

Our storage layer must make a distinction between two types of data that can be stored in the system:

1. Immutable, self-verifying data. For instance, CFS data blocks do not change over time and are indexed by the hash of its contents, and therefore their integrity can be verified.
2. Mutable mappings that may be subject to a replay attack; CFS root blocks are an example. For such data we need to make sure we are retrieving the latest version. We do expect this data to be signed so that its integrity can also be verified by clients.

To ensure freshness of the second type of data, the client must extend it with a monotonically increasing version number that is also covered by the signature.

We will now present the algorithms that are involved in the operations of the storage layer. We will begin by describing how we store and retrieve data when the configuration does not change, and then describe what happens during configuration changes — how state transfer takes place and how the storage and retrieval algorithms work in the presence of a changing configuration.

5.2.1 Storage and Retrieval with a Static Configuration

In general, the storage algorithms for both types of data must ensure that $2f + 1$ replicas claim to have stored the data. The reason why we do not wait for the remaining f replicas is that they may be faulty and are never going to respond. Therefore, we need to make progress after hearing from only $n - f = 2f + 1$ replicas.

Thus for a client to write data of either type, it must send the write to at least $2f + 1$ replicas. It needs to receive acknowledgments from $2f+1$ replicas before it can be certain that the write succeeded.

In all storage operations, the integrity of the data should be verified by the replicas storing that data item to avoid garbage being stored. For data of type (1), this means verifying that the ID is a hash of the content.

Reading data of type (1) is simple: it is sufficient to read from one replica, as long as the client verifies that the hash of the contents matches the ID. If it does not, the client repeats the read but this time asks for all the replicas of the data until it finds one that matches.

Version numbers for writes of data of type (2) are generated by the clients, and storage nodes reject blocks with version numbers not greater than the current version that is stored. The client doing the write can simply generate this version number (through prior knowledge or storage outside the system). Or, it can read the current version to learn the current version number; then it can increment that number to obtain the new version number and use it when writing the data.

Reads for data of type (2) have to wait for $2f + 1$ replies and choose the block with the highest version number. Note that reading from $2f + 1$ replicas has the nice property that the set of replicas we read from intersects the set of replicas for the last write in at least one non-faulty replica. This can

be seen as a particular instance of a dissemination Byzantine quorum system [9].

When reading data of type (2), clients must ensure they are receiving data from the correct nodes, since otherwise they could be tricked into a replay attack where they would be presented out-of-date versions of the data. Therefore, requests for data of this type contain a nonce that is sent with the signed reply to ensure its authenticity and freshness. As mentioned, clients know the public keys of P2P nodes since this is part of the configuration information.

One problematic situation is a faulty client sending different data items of type (2) to be stored at different replicas with the same version number. We address this issue by allowing temporary inconsistencies (different reads may obtain different values for the data item in question after the faulty write) which will be solved when a non-faulty client overwrites that data item. This temporary inconsistency might be avoided by having the replicas perform a consensus on the value being stored, but we decided against this because of the slowdown it imposes, especially in a wide area network.

These algorithms assume, like CFS does, that there are no concurrent writes to the same data item. Serializability for concurrent data accesses can be implemented by ordering the writes by version number and, if the version numbers are the same, by the hash of the contents.

5.2.2 State Transfer

When the configuration changes, the P2P nodes that are responsible for storing a particular data item in the new configuration may differ from those storing it in the previous configuration. In this case, state transfer must take place to bring the new replicas for that data item up-to-date.

This task can be simplified if all replicas involved in this process have access to both the old and the new configurations. This way everyone knows which nodes contain data they need, and which nodes need data they have.

So assume that both configurations are distributed to all nodes (at different times, though). When a node receives a new configuration ($n+1$), and realizes it is now responsible for some new data items, it reads that data from the nodes that held it in configuration n. Nodes receiving data from configuration n should check the validity of the sender, the integrity of the data, and, if it is of type (2), should merge the copies it receives from different replicas (i.e., keep the one with the latest version number). Note that for data of type (1) it is enough to receive the data from one replica, whereas for data of type (2) the node must receive data from $2f + 1$ replicas and pick the one with the highest version number.

We also assume that nodes include identifiers of the latest configuration they know in the messages they exchange. If a node detects an identifier greater than its current configuration, it asks the node that sent the identifier for a copy of that configuration. The same applies to clients: if a client tries to store a data item and a storage node that it talks to knows a configuration later than the one the client is using, it forces the client to fetch the new configuration, and the client repeats the store operation in the new configuration. This prevents different clients from performing concurrent reads and writes in different configurations, which could lead to inconsistencies.

6 Conclusions

In this paper we presented techniques to build Byzantine fault-tolerant peer-to-peer systems. To achieve this level of robustness in peer-to-peer systems we needed to deviate from traditional peer-to-peer architectures in two fundamental ways. First, we proposed a hybrid system, with symmetric storage nodes that implement the peer-to-peer system and a configuration service that performs admission control and determines the current configuration. Second, the P2P nodes (and also the CS nodes) must be server machines dedicated to run the storage application, and not client machines that run arbitrary code and are constantly entering and leaving the system. Our server machines have secure co-processors and perform proactive recovery.

We also sketched the design of a system built according to these principles. We believe that this design will work and the resulting system should perform well. The P2P nodes will not be burdened with determining membership and therefore can perform their storage function without interference except during configuration changes, which occur infrequently. In addition, because the P2P nodes and the clients store the entire configuration, routing is very efficient.

There is much future work that needs to be done, however. There are still many design issues to be resolved, and then we need to implement the system. We also plan to study the use of the system in various applications, and to extend it as needed. For example, one needed extension is providing support for a delete operation and a quota system to limit the amount of storage a client can use. The two go together, because when a quota system is deployed, a client trying to write to the system when its quota is reached needs to have some way of freeing up space to be able to continue using the system. We are working on algorithms to support these two features.

Acknowledgements

We would like to thank Emil Sit, Steven Richman, Chuang-Hue Moh, Sameer Ajmani, and the anonymous referees for their helpful comments on drafts of this paper.

References

[1] Miguel Castro and Barbara Liskov. Practical Byzantine fault tolerance. In *Proceedings of the 3rd Symposium on Operating Systems Design and Implementation (OSDI-99)*, pages 173–186, New Orleans, Louisiana, February 1999.

[2] Miguel Castro and Barbara Liskov. Proactive recovery in a Byzantine-fault-tolerant system. In *Proceedings of the 4th Symposium on Operating Systems Design and Implementation (OSDI-00)*, pages 273–288, San Diego, California, October 2000.

[3] Frank Dabek, M. Frans Kaashoek, David Karger, Robert Morris, and Ion Stoica. Wide-area cooperative storage with CFS. In *Proceedings of the 18th ACM Symposium on Operating Systems Principles (SOSP-01)*, pages 202–215, Banff, Canada, October 2001.

[4] Alan Demers, Dan Greene, Carl Hauser, Wes Irish, John Larson, Scott Shenker, Howard Sturgis, Dan Swinehart, and Doug Terry. Epidemic algorithms for replicated database maintenance. In *Proceedings of the 6th Annual ACM Symposium on Principles of Distributed Computing*, pages 1–12, Vancouver, Canada, August 1987.

[5] John Douceur. The Sybil attack. In *Proceedings of the 1st International Workshop on Peer-to-Peer Systems (IPTPS '02)*, Cambridge, Massachusetts, March 2002.

[6] Freenet. http://freenet.sourceforge.net/, 2002.

[7] Gnutella. http://gnutella.wego.com/, 2002.

[8] David Karger, Eric Lehman, Tom Leighton, Matthew Levine, Daniel Lewin, and Rina Panigrahy. Consistent hashing and random trees: Distributed caching protocols for relieving hot spots on the World Wide Web. In *Proceedings of the 29th ACM Symposium on Theory of Computing*, pages 654–663, El Paso, Texas, May 1997.

[9] Dahlia Malkhi and Michael Reiter. Byzantine quorum systems. In *Proceedings of the 29th ACM Symposium on Theory of Computing*, pages 569–578, El Paso, Texas, May 1997.

[10] Ralph C. Merkle. A Digital Signature Based on a Conventional Encryption Function. In Carl Pomerance, editor, *Advances in Cryptology - Crypto'87*, number 293 in Lecture Notes in Computer Science, pages 369–378. Springer-Verlag, 1987.

[11] David L. Mills. Network Time Protocol Specification, Implementation and Analysis. Network Working Report RFC 1305, March 1992.

[12] Sylvia Ratnasamy, Paul Francis, Mark Handley, Richard Karp, and Scott Shenker. A scalable content-addressable network. In *Proceedings of the ACM SIGCOMM 2001 Conference (SIGCOMM-01)*, pages 161–172, San Diego, California, August 2001.

[13] Rodrigo Rodrigues, Miguel Castro, and Barbara Liskov. BASE: Using abstraction to improve fault tolerance. In *Proceedings of the 18th ACM Symposium on Operating System Principles (SOSP-01)*, pages 15–28, Banff, Canada, October 2001.

[14] Antony Rowstron and Peter Druschel. Pastry: Scalable, decentralized object location and routing for large-scale peer-to-peer systems. In *Proceedings of the 18th IFIP/ACM International Conference on Distributed Systems Platforms (Middleware)*, pages 329–350, Heidelberg, Germany, November 2001.

[15] Antony Rowstron and Peter Druschel. Storage management and caching in PAST, a large-scale, persistent peer-to-peer storage utility. In *Proceedings of the 18th ACM Symposium on Operating Systems Principles (SOSP-01)*, pages 188–201, Banff, Canada, October 2001.

[16] Secure Hash Standard. US Dept. of Commerce/NIST, 1995.

[17] Emil Sit and Robert Morris. Security considerations for peer-to-peer distributed hash tables. In *Proceedings of the 1st International Workshop on Peer-to-Peer Systems (IPTPS '02)*, Cambridge, Massachusetts, March 2002.

[18] Ion Stoica, Robert Morris, David Karger, Frans Kaashoek, and Hari Balakrishnan. Chord: A scalable Peer-To-Peer lookup service for internet applications. In *Proceedings of the ACM SIGCOMM 2001 Conference (SIGCOMM-01)*, pages 149–160, San Diego, California, August 2001.

[19] Ben Y. Zhao, John Kubiatowicz, and Anthony D. Joseph. Tapestry: An infrastructure for fault-resilient wide-area location and routing. Technical Report UCB/CSD-01-1141, Computer Science Division, University of California at Berkeley, April 2001.

[20] Lidong Zhou, Fred B. Schneider, and Robbert van Renesse. COCA: A secure distributed on-line certification authority. Technical report, 2000-1828, Department of Computer Science, Cornell University, December 2000.