Scalable State Replication with Weak Consistency

Svend Frølund[1] Vana Kalogeraki[2] Fernando Pedone[1,3] Jim Pruyne[1]
[1]Hewlett-Packard Laboratories (HP Labs), Palo Alto, CA 94304, USA
[2]University of California–Riverside, Riverside, CA 92521, USA
[3]Ecole Polytechnique Fédérale de Lausanne (EPFL), CH-1015 Lausanne, Switzerland

Abstract

Initial work on peer-to-peer systems has focused on finding *information in large-scale decentralized systems. More recently, the focus has shifted to* sharing *information in such contexts. Meeting this goal in environments in which many data replicas change their state frequently is very challenging. The objectives of the work described in this paper is two-fold: designing mechanisms that allow information lookup based not only on unique data keys, but also on meta data, and enabling efficient and scalable implementation of data sharing by providing a notion of consistency weaker than existent proposals. The paper formalizes the notion of weak consistency in peer-to-peer environments, and presents detailed implementations of our information lookup and data sharing mechanisms.*

1. Introduction

The first wave of peer-to-peer systems, such as Napster [3] and Gnutella [2], focussed on *finding* information in large-scale, decentralized systems with an ad-hoc structure. More recently, peer-to-peer systems, such as OceanStore [9], are starting to address the more ambitious issue of *sharing* information in a large-scale, decentralized setting. Systems that address information sharing typically have to deal with information that changes over time. In a decentralized setting, this means dealing with consistency of mutable, replicated data. Implementing consistency for a large number of mutable data replicas in a scalable manner is very challenging. Our goal is to provide middleware building blocks that meet this challenge in a peer-to-peer setting.

As observed by other researchers, such as [11, 19, 14], the notion of a distributed hash table is a nice middleware building block to enable information sharing. We can use a hash table as an information directory. The values stored in a hash table can either be the information itself or some reference to the information. The hash table keys can be used to name and access the information. The information associated with a given name can now change over time: we can update the key-to-value mapping in a hash table. In short, peers can share information through read and write operations on a distributed hash table.

We extend existing work on hash-table-based middleware in two directions:

- Where hash-tables support information lookup based on unique keys only, we provide information lookup based on general queries over arbitrary information meta-data.
- Where traditional implementations of distributed hash-tables seek to provide the usual, single-copy semantics for updates and lookups, we define and implement a weaker notion of read-write consistency that allows for a more distributed and scalable system.

In our system, information meta-data is a set of arbitrary name-value pairs that describe a piece of shared information. For example, if the shared information is a document, the meta-data may contain the name of the author and a number of keywords for the document content. If the shared information is an enterprise directory, each entry in the directory may have meta data that contains the location code and reporting structure for the employee in question. Because of the more flexible lookup capabilities, we refer to our system as a *state manager* rather than a hash-table. A state manager maintains a set of *variables*, where each variable maps meta-data to a piece of shared information. The basic operations on a state manager is the creation and deletion of variables, the lookup of information based on a meta-data query, and the assignment of a new piece of information to an existing variable.

Being able to access information based on arbitrary meta-data makes the system more flexible. In fact, the reliance in existing systems on unique keys has already been identified as a weakness [15]. However, being able to access information based on meta-data also introduces new challenges. For example, existing systems that implement a distributed hash-table typically partition the key space among the nodes of the system, and define efficient routing algorithms through the network. In this way, both lookup and update operations can be routed to the node responsible for maintaining the mapping for a given key. Because a state manager has arbitrary meta-data, we cannot partition the variable space and route each state manager operation to a single node only: we do not know, a priori, which nodes will satisfy a given meta-data query.

In terms of consistency, existing work seeks to implement the "usual" hash-table semantics: a distributed hash table behaves as if it were a centralized data structure that is accessed sequentially. It is very hard to efficiently implement the usual hash-table semantics in a very large-scale distributed system. As an alternative, we define and implement a weaker notion of consistency that to some extent exposes the presence of a replicated state manager.

Our notion of consistency reflects the semantics typically associated with existing large-scale information-sharing systems, such as the Domain Name System (DNS), the World-Wide Web, and wide-area publish-subscribe systems. In these systems, update propagation is not instantaneous, but rather time bounded. That is, users of these systems may observe information being out-of-date for some bounded period of time. Furthermore, a given information copy may not necessarily receive all updates. Reception of the latest version makes all older versions obsolete and unnecessary.

We have implemented a distributed, decentralized state manager that provides our weak consistency model. We decompose the general state manager into variable management and discovery sub-systems. This has allowed us to focus on specific problems and use existing work where appropriate. The discovery capability is built on an enhanced gnutella search network which interoperates with the existing network, but scales well and allows arbitrary meta-data queries to be performed. We have designed new protocols for the formation of networks to update and propagate changes to state manager variables. In both cases, we retain the benefits of peer-based systems in the areas of scalability, reliability and adaptation to change.

The rest of the paper is structured in the following manner. We formalize out notion of weak consistency in Section 2. We present our implementation in Section 3. We discuss possible improvements to the implementation in Section 4. Section 5 outlines related work, and Section 6 provides concluding remarks.

2. System Model and Correctness

2.1. Processes and Communication

We consider a system with n processes and use Π to refer to the set of processes ($\Pi = \{p_1, p_2, \ldots p_n\}$). We also refer to the elements of Π as peers, and use the terms "process" and "peer' inter-changeably throughout this paper. At any given time, a process is either *present* or *absent*. When a process is present, it executes at its own speed and according to its prescribed algorithm. When a process is absent, it is completely passive and does not execute any instructions. In particular, we assume that a proces never acts maliciously.

The transition from present to absent, and vice versa, is atomic. A *leave* event at a process p captures the transition of p from present to abssent; a *join* event captures the transition from absent to present. A leave event may happen because a process crashes or it may happen because the process chooses to disconnect from the network. Processes may leave silently, we do not assume that a processes notifies other processes that it is about to leave. A join event may happen because a process recovers from a crash, or because it chooses to connect to the network. A process may either be absent or present initially.

A process is called *unstable* if it leaves and joins and infinite number of times. A process is called *correct* if there is a time after which it is permanently present.

Processes communicate by passing messages. We assume that communication links can lose messages but not corrupt them. Moreover, if a process p keeps sending some message m to process q, and q remains present, then q eventually receives m. Therefore, processes can build reliable communication on top of unreliable links by periodically resending messages. Our protocols do not always need reliable communication, however, and so, unless stated otherwise, we consider that communication is unreliable.

2.2. Correctness

We define what it means for a state manager to be correct. We focus on the notion of consistency that a state manager must ensure relative to queries and updates of its variables. To formally define our notion of single-variable consistency, we model a variable as a special kind of read-write register that we refer to as a *convergence register*. In defining consistency, we do not consider the creation or deletion of variables, nor do we capture the ability to perform meta-data queries. Furthermore, we define the properties of a single register only—a state manager does not provide consistency guarantees for multi-variable operations.

A convergence register has two operations: read and write. A write operation takes a value in a domain Value ($v \in \mathsf{Value}$). A read operation returns a value in $\mathsf{Value} \cup \{\bot\}$ ($\bot \notin \mathsf{Value}$). Informally, write assigns a new value to a variable and read determines the current value of a variable.

To distinguish between different invocations of the write operation, we assume that each value is written at most once. In practice, we can "implement" this assumption by associating a unique identifier with each invocation of the write operation and consider the identifier to be part of the written value.

We specify the properties of a convergence register in terms of events. In modeling a system's behavior, we consider communication events and register events. Communication events reflect message passing between processes. The event $\mathsf{send}_p(m)$ occurs when a process p sends a message m, and the event $\mathsf{receive}_q(m)$ occurs when a process q receives a message m.

Register events reflect the start and completion of register operations (read and write). The event sread_p captures the start of a read operation at a process p. A start event sread_p occurs when p invokes the read operation on a register. The event $\mathsf{eread}_p(v)$ denotes the completion of a read operation that returns v at a process p. A completion event $\mathsf{eread}_p(v)$ occurs when a process p returns from the invocation of a read operation. Similarly, the events $\mathsf{swrite}_p(v)$ and $\mathsf{ewrite}_p(v)$ denote the start and completion of a write operation with parameter v at a process p, and these events occur when p invokes a write operation or when p returns from such an invocation.

We assume that the events at any given process p are totally ordered according to a precedence relation $\rightarrow_p$. We use the inherent ordering of message-passing events to extend this local precedence relation to a global precendece relation $\rightarrow$. Where $\rightarrow_p$ is a total order, $\rightarrow$ is a partial order only. In accordance with well-established practice [10], we define $\rightarrow$ in the following manner:

1. Sending a message always preceedes receiving it:

   $\mathsf{send}_p(m) \rightarrow \mathsf{receive}_q(m)$

2. The global precedence order obeys the per-process total order:

   $e_1 \rightarrow_p e_2 \Rightarrow e_1 \rightarrow e_2$

3. The global precedence order is transitive:

   $e_1 \rightarrow e_2 \wedge e_2 \rightarrow e_3 \Rightarrow e_1 \rightarrow e_3$

We extend the notion of precedence to written values in the following way: if $\mathsf{ewrite}_q(v') \rightarrow \mathsf{swrite}_p(v)$, we say that v' *preceedes* v. Moreover, we assume that $\perp$ preceedes any written value.

If a run contains the event $\mathsf{eread}_p(v)$, we say that p *reads* v. Similarly, if a run contains $\mathsf{ewrite}_p(v)$, we say that p *writes* v. If $\mathsf{eread}_p(v) \rightarrow \mathsf{eread}_p(v')$ we say that p reads v *before* it reads v'.

With this terminology, we can now define the properties of a convergence register. In the properties, v, v_1, and v_2 are all non-$\perp$ values.

- TERMINATION: If a process invokes read or write, and then remains present, then the invocation eventually returns.

- INTEGRITY: If a process reads v then some process invokes write with parameter v.

- ORDER: If a process reads v_1 before it reads v_2, then no process reads v_2 before it reads v_1.

- PROGRESS: If a process writes v, and then remains present, then eventually no correct process reads a value that preceedes v.

- CONVERGENCE: If a run contains only a finite number of swrite events, then eventually each read operation at every correct process returns the same value.

The ORDER, PROGRESS, and CONVERGENCE properties complement each other to define a weak, yet useful, notion of consistency for replicated state. Without order, the processes may disagree about the current value for arbitrary periods of time. For example, if two processes, `p1` and `p2` concurrenctly write the values `v1` and `v2`, `p1` may initially read `v1`, and then later read `v2`, whereas `p2` may initially read `v2`, and then later read `v1`. To satisfy convergence, the processes must eventually agree on a common value, but this agreement can be arbitrarily postponed (the agreement is only required to hold for correct processes).

Without the PROGRESS property, a register may forever use the first value written as the return value from all read operations. That is, the register implementation may simply drop all subsequent values written.

Without the CONVERGENCE property, two parts of the network may forever operate independently, that is, processes are allowed to perpetually read and write values in isolation. Roughly speaking, convergence supplements progress to ensure some notion of "agreement" in a system with concurrent write operations. If write operations are totally ordered according to the precedence relation, and if processes that invoke write are perpetually present, progress implies convergence.

3. Implementation

Our algorithmic approach to the state management abstraction is to decompose the problem into two sub-problems. The first is discovery. That is, how a new comer to the network is able to discover a variable of interest and attach to the network. The second is managing state updates and providing the consistency guarantees defined in the previous section.

For discovery, our approach is to augment an existing search network, gnutella [2]. Gnutella's failures in the area of scalability have been well documented [4]. However, rather than replace the already defined protocols, we take the approach of performing more intelligent peer selection algorithms at each process. The goal is to generate a search network which is more efficient than is found in today's randomly generated networks.

State updates are handled by designating one process as the *owner* (process $S4$ in the example network) of a state variable at any point in time. Only the owner may update the state of the variable by adding a new attribute or changing the value of an attribute. The updated state is periodically propagated to other processes which have previously shown interest in the state of the variable. These two algorithms are described in more detail in the following sections. The algorithms as described here are what have been implemented in our prototype. Specific extensions to the implemented protocols are discussed in the following section.

3.1. Intelligent Peer Searching

The network of processes can easily grow to a large number of participants with processes activating or disappearing dynamically. As a result, the topology of the network can be random with various connectivity degrees between the peers, regardless of the physical location, resource capabilities or the state of the processes. To address this problem, our approach is to make intelligent connections among the peers. This has the important advantages: (1) reduces the number of messages in the network and also the number of peers that process and propagate these messages and, (2) scales well with respect to the size of the network.

3.1.1 Discovering the Network of Processes

A process joins the network of processes by establishing a connection with at least one process currently in the network. These are peers with which the process had previous connections to and is likely to connect to in the future. At any given time a process can become absent, in which case the process will have to find an alternative peer to connect to or probes a centralized server such as [3] to obtain a list of currently present processes in the network. The process constructs and maintains a list of known processes, called `PeerList`. Each process has a limited amount of bandwidth in the network and therefore limits the number of connections it can accept. The process of discovering the network of processes, is facilitated by the following two communication events:

- `ping:` discover active processes in the network
- `pong:` a process that receives the ping message accepts the connection by replying with a *pong* message. In addition, it piggybacks information about the process, such as its geographic location and the type of connection (e.g., modem, DSL) it supports.

3.1.2 Intelligent Peer Connections

A process searches in the network by sending *query* messages to its peer processes. A query message contains a constraint that is evaluated locally at each peer to determine what results to return. The constraint includes a set of meta-data that describe the search request.

The process that receives the query message evaluates the query locally and should a number of results be found, replies with a *query_hit* message that includes an enumeration of the matching search results along with the IP address of how they can be invoked.

The problem in the current pure P2P networks such as Gnutella [2] is that because the networks are formed in an arbitrary manner the messages may have to travel a large number of hops from one process to another until the results are found. To address this problem, our algorithm selects the connections among peers so that peers with similar interests (such as same file replicas) are connected to each other. This has the important advantage of reducing the end-to-end delay in finding the data and minimizing the number of messages in the network [13].

To identify good peers, each process builds a profile of its peer processes. The profile includes the list of the most recent K past queries and the specific peer that provided the answer for. The process accumulates the list of past queries by two different mechanisms: (1) by monitoring the number and type of requests (Query messages) performed by this process in the network, and (2) by recording the peer processes that replied to its requests (QueryHit messages) and the number and type of requests they generated.

The process updates the list of queries at its local repository. Note though, that for each process peer this list is incomplete, because these are only the queries that were routed through this node. As the system operates, the size and accuracy of the list increases over time.

When a process identifies that a peer is frequently producing good results to its requests, it moves closer to it in the network by establishing a direct connection to that peer. If the number of available connections at the node exceeds, the process removes one of its existing connections. The decision of which connection to remove depends on (1) how many results are generated from that peer, (2) how many results are propagated from remote peer processes and (3) the period of connection time. Based on the processes' behavior, the same peer processes may have different importance for different periods of time. The pseudocode for selecting good connections, executed at each peer, is illustrated in figure 1.

3.2. State Update and Propagation

3.2.1 Joining the network

The state update algorithm is intended to provide the consistency properties described in Section 2. The first step in the system is for a process to discover at least one other process that has a value for the desired state variable. It uses the discovery protocol described in Section 3.1 to do this. When a remote process is found, the local process sends a `VARIABLE_REQUEST` message to it. The remote process responds by sending a `VARIABLE_UPDATE` message back containing its current value for the state of the variable. In addition, the remote process and the local process add the other to its list

```
intelligentConnections() {
  wait for incoming msg
     if received QUERY msg(search constraints)
        evaluate query at the local repository
        if (successful)
            create QUERY_HIT msg(IP_addr, port, replies)
            forward QUERY_HIT msg to the requesting node
        propagate QUERY msg to all other
          direct peers except requesting
     if received QUERY_HIT msg(IP_addr, port, replies)
        update replies for peer IP_addr in PeerList
        if number of replies from indirect peer is high
           connect to indirect peer
           if number of available connections exceeds
              remove direct peer with less replies
}
```

Figure 1. Intelligent Connections pseudo-code.

```
boolean requestOwnership(Variable) {
  try
    create reliable channel endpoint
    loop upto retry limit
      create REQUEST_OWNERSHIP
        msg(channel address, Variable.Id)
      send msg to Variable.Owner or
        broadcast to all peers if too many retries
      wait for connection on channel
      if (connected before timeout)
        receive VARIABLE_UPDATE with current state
        close channel
        set Variable.Owner to self
        return true
    end loop
  catch any faults
  return false
}
```

Figure 2. Ownership request pseudo-code

of peers for that variable. This per-variable peer list is used in propagating state updates as will be described below. By joining the peer list, we have essentially joined a process from the search network into the state update network.

3.2.2 Performing an update

A process that wishes to update the state of a variable must first be granted `ownership` of the variable. Only the present owner of a variable is allowed to perform updates. Every variable contains an attribute containing the network address of the current belief about the owner. An owning process can simply perform the update locally with no interaction with other processes. We believe this will be the common case as most applications will have a "locality of updates" in which updates will be performed at one location most of the time. In section 4 we discuss extensions that eliminate the single owner as a point of failure which would make further updates to a variable impossible.

A non-owner process that wishes to perform an update must for be granted ownership. This done using the algorithm

```
processRequestOwnershipMsg(Variable) {
  try
    if (Variable.Owner == self &&
         willing to grant ownership)
      create VARIABLE_UPDATE msg
        containing state of the variable
      connect to endpoint provided in the msg.
      send VARIABLE_UPDATE msg on channel
      close channel
      set Variable.Owner to sender of msg.
    else if (Variable.Owner != self)
      forward msg to Variable.Owner
  catch any faults
}
```

Figure 3. Ownership request message processing

shown in Figure 2. The process starts by creating a reliable channel on which ownership will be granted. We use a reliable channel in this case to help ensure transfer of ownership. Next, it creates a `REQUEST_OWNERSHIP` message with the address for the channel and the identity of the variable it is requesting ownership of. This message is sent to the node's current owner value for that variable. When a connection is received on the channel, it reads the current state of the variable which is sent by the current owner, closes the channel, and sets itself as the owner. If the connection is not received in a timely manner, the `REQUEST_OWNERSHIP` message is re-sent. Multiple failures likely indicate an owner that is no longer on the network, or a network failure between the process and the owner, so the message can be broadcast to all of the variable's peers to help ensure that the current owner is found. If faults are detected at any point or too many messages have been sent, we assume that a successful ownership transfer has not occured, so we leave the owner set to our original value. This would mean that the original update operation would been considered to have failed, though a higher-level application would be free to re-initiate the update if it wanted to.

The algorithm for the receiver of the `REQUEST_OWNERSHIP` is shown in Figure 3. We first must be sure that receiving process is the current owner, and that it believes granting ownership to be a good idea. Choosing not to grant ownership is a heuristic decision, but may be based on frequent updates at this process so there's an expectation that it will need to regain ownership soon. By maintining ownership, it can avoid ownership moving too frequently. If it is willing to grant ownership, it simply creates a `VARIABLE_UPDATE` message containing the state of the variable, connects to the requester on the reliable channel it created, and sends the message on this channel. It finally closes the channel and assumes the the requester is now the current owner. As above, if any failures in the communication are detected, we assume ownership has not

```
processVariableUpdateMsg(VariableUpdate) {
  find local variable with id match-
ing the id in the msg.
  if (VariableUpdate update count > local update count)
    copy attributes from VariableUpdate
      into local copy of Variable
    forward msg. to all peers for this
      variable except sender
  add sender of the msg. to peer list for this variable
}
```

Figure 4. Variable update message processing

been succesfully transfered, so the process retains ownership.

Because of the lax consistency model provided by the system or because a desperate process may broadcast its ownership request, it is possible that a REQUEST_OWNERSHIP message could be received at a non-owner process. In this case, the receiving process simply forwards the REQUEST_OWNERSHIP message to its current owner attribute value on behalf of the requesting process. This process can continue until it is received at the current owner. We refer to this as "chasing the owner." Similar approaches have been used in the past for communicating with migrating objects [8].

There are cases, however, where ownership transfer can be dynamic enough that the chase will never complete. The retries and broadcast fall-back by the requester are intended to reduce the likelihood of this occuring. These retries and broadcast are safe because the ownership request operation is guaranteed to be idempotent because the requester will accept only one connection on the reliable channel it establishes for the transfer of ownership. Further connection requests will simply fail on the owner side, and no update to the owner value will be performed.

3.2.3 Propagating updates

Periodically, the owner of each variable sends the current state of the variable to each of the processes on its peer list. It first increments an update count attribute and then sends the contents of all changed attributes since the last update, including the update count, in a VARIABLE_UPDATE message. We perform this operation only periodically, rather than as each update occurs, with the belief that updates will be clustered temporally. By waiting until the interval expires all of the updates can be sent at one time rather than triggering a rapid sequence of updates. If no updates occur during the interval, we do not need to send the update. However, because of the unreliable channels we do send updates containing all attributes after multiple intervals of inactivity to help ensure that the current state is received by all processes.

When a process receives a VARIABLE_UPDATE it compares the update count with the update count in its local copy as shown in Figure 4. If the value in the received message is greater, it copies the attributes from the message into its local copy. If the count is lower, the process assumes that this message was somehow delayed, and contains stale data, so the message is simply ignored. An equal value should never be possible because of the duplicate message elimination performed at the lower message processing layers.

An accepted message is also forwarded the message to all of its peers for this variable with the exception of the source of the message. In this way, the new state is flooded through the network in the same way that query messages are propagated in gnutella and other file sharing networks. We believe that this method is appropriate here for a number of reasons:

1. The number of processes interested in a single variable will be small relative to the total number of processes in the overall network, so the scalability requirement is less stringent.

2. The frequency of updates is small compared to the frequency that queries are generated in the general search network. Further, we can tune the update periodicity if update traffic becomes too high.

3. More sophisticated network generation algorithms, similar to the intelligent peer, could be created if needed.

Finally, we add the sender of the message to the local peer list for this variable. Because we've received this message, we know this peer is interested in this variable. Due to ownership transfers, it may become possible that the local process will be closer to a new owner, so it would have to reverse roles with the sender of the message, and would be responsible for propagating updates to it. In the appendix, we provide justification for why our algorithm is correct relative to the properties defined in section 2.2.

4. Fault-Tolerant State Management

4.1. Group Communication Primitives

We replicate the variable owner to tolerate failures and use Group Communication primitivies to ensure consistency. In a Group Communication system, processes are organized in groups. Processes in a group use communication primitives with well-defined semantics in case of failures to communicate with each other. Furthermore, there exist mechanisms that regulate how processes join and leave a group, as we now explain.

Atomic Broadcast is a Group Communication primitive that provides *atomicity* and *total order* [7]. Let m and m' be two messages broadcast in some group g of processes. The atomicity property guarantees that if a member of g delivers m (resp. m'), then every correct member of g also delivers m (resp. m'). The total order property guarantees that no two members of g deliver m and m' in different orders.

Virtual Synchronous Systems accommodate processes joining and leaving a group [5, 6]. It is defined in the context of a group g and is based on the notion of a *sequence of views* $v_0(g), v_1(g), \ldots, v_i(g), \ldots$ of group g. Each view defines the composition of the group at some time t, that is, the processes that are believed to be the correct members of the group at time t. Whenever a process wants to join the group, leave the group, or is suspected to have failed, a new view is installed to reflect the new membership.

Roughly speaking, if a process member of group g currently in view $v_i(g)$ broadcasts a message m, Virtual Synchronous Systems guarantee that if some process p in $v_i(g)$ delivers m before installing view $v_{i+1}(g)$, then no process installs view $v_{i+1}(g)$ before having first delivered m.

4.2. Handling Failures Consistently

Our approach consists in associating to each variable a group of owners, instead of a single owner. Thus, each variable is associated to a fixed group, whose identity is the variable name. Process joins and leaves are handled by the Group Communication system, as described in the previous section. Likewise, processes suspected to have failed are automatically excluded from the group by the group membership mechanism. We describe a simple mechanism to handle variable updates.

Only processes belonging to some variable group can perform updates to that variable. Thus, if a process that does not belong to the group variable wants to modify the variable, it has first to join the group. To join a group, a process has to find another process that is currently member of the group and then send it a request. Finding such a process can be done with a mechanism similar to the one described in Section 3.2 to find the current owner of a variable. Notice that this join operation is performed after the process has already joined the network.

Since several processes may simultaneously decide to update the state of a variable, care must be taken so that the variable state remains consistent after the operations have been executed. To guarantee a correct execution we require each process to broadcast its update to all the members of the variable group, that is, variable updates are performed using *active replication* [18]. Active replication requires that each process execute the request deterministically, so that they will all end up in the same state after the execution of the operation. Since the operations in our case are simple requests to change the state of a local variable, determinism is easily enforced.

Furthermore, as before, updated variable states have to be forwarded to the other processes in the system. To make sure that an old update does not overwrites a more recent one, timestamps are used. So, besides executing the update locally, processes in a group also compute the update timestamp. Computing a local timestamp can be easily done deterministically. Once the update timestamp has been calculated, the process in the variable group forwards it to each of the processes in its process list.

A process does not forward the update to other processes in its group, since from the Atomic Broadcast properties, unless such processes crash, they will eventually receive the operation. Processes that do not belong to the variable group will receive, possibly more than once, the new variable state and perform the update locally. Read operations do not incur in any network traffic: the current value of the variable is simple returned. As in the previous section, the appendix provides a discussion of correctness for the augmented algorithm presented here.

5. Related Work

Peer-to-peer systems have received much attention recently for sharing information among a large number of participants. Despite their immense popularity, systems such as Napster[3], Freenet [1] and Gnutella [2], fail to provide consistency, availability or scalability guarantees to the distributed applications.

This first generation of peer-to-peer systems has motivated the development of more robust solutions for sharing information in a large-scale, decentralized environment.

The Oceanstore system [9] is a utility infrastructure designed to provide secure, highly available access to persistent objects in a large-scale environment. Its main attractive features include persistent storage, fault tolerance and support for nomadic data in a fundamentally untrusted environment. They focus primarily on ensuring atomic operations on the data objects; changes to objects are made by client-generated updates and concurrent updates are handled through a conflict resolution scheme. In this regard, they employ primary and secondary tiers of replicas and use a Byzantine agreement protocol to choose the final commit order for the updates.

The Past system [17] is an Internet based global-scale storage utility that relies on the notion of immutable data objects. Its main goals are strong persistence, high availability, scalability and security. The key idea is that storage nodes and files are assigned uniformly distributed identifiers, and replicas of the files are stored at the nodes whose identifiers are numerically closest to the file's id. In contrast to our approach, Past is not intended to address the problem of changing data. Rather, it aims at providing efficient request routing, deterministic object location and load balancing.

The Farsire project [20] buils a servless, distributed file system that exploits the underutilized storage and communication resources of a large organization. Its distinguish characteristic is security; it can be employed on an existing desktop infrastructure without assuming careful administration or mutual trust among the client machines. The system exhibits location-transparent access to private and public files, security and resistence to Byzantine threats, self-configuration and adaptability. It is not clear however, how well it scales in a wide-area environment.

Peer-to-peer systems have inspired work on algorithms for efficient location and routing [19, 14, 16]. These are dictated by a consistent mapping between an object key and a hosting node. Locating an object is reduced to the problem of routing the request to the node responsible for storing the object's key. Similar to our objective, their goal is to minimize the number of logical hops that need to be traversed to locate an object. The difference with our work is that we assume that the topology of the network is unknown, efficient routing is handled by intelligently manipulating the connections among the peers.

The problems faced today by the peer-to-peer systems are similar to issues addressed by existing directory services such as the Internet Domain Name System (DNS) [12]. DNS uses a hierarchical namespace where each domain has authority for serving parts of the DNS database. Similar to our approach, DNS uses a weak consistency model for updating replicated information. DNS hosts can query each other and cooperate by propagating data requests across the network.

6. Conclusions and Future Work

This paper describes the properties and implementation of a decentralized state manager we have developed. We decompose our system into two modules: a state propagation module and a discovery module. This modular design allows us to focus on specific problems and use existing work where appropriate. Our state propagation algorithm provides a consistency guarantee which reflects the semantics typically associated with existing large-scale information-sharing systems, such as the Domain Name System (DNS), the World-Wide Web, and wide-area publish-subscribe systems. Our notion of consistency, while strong enough to be useful, allows an efficient implementation. The discovery capability extends Gnutella-like search network mechanisms to allow better scalability and arbitrary meta-data queries.

The state management system and the consistency notions it provides are intended to be building blocks for other applications. We are developing two applications on top of our prototype. One provides file system functionality and the other supports e-mail type messaging. In both cases, the applications support standard protocols, WebDAV and IMAP respectively, to allow them to interoperate with off-the-shelf clients. We believe that this approach will allow us to use the system effectively, and validate our beliefs about the usefulness of our consistency model and the efficiency of our implementations. We also intend to augment the system with the group communication schemes described to eliminate the single point of failure in the current approach.

References

[1] The freenet home page, 2000. freenet.sourceforge.net.

[2] The gnutella home page, 2000. gnutella.wego.com.

[3] The napster home page, 2000. www.napster.com.

[4] E. Adar and B. Huberman. Free riding on gnutella. Technical report, Xerox PARC, Aug. 2000.

[5] K. Birman, A. Schiper, and P. Stephenson. Lightweight causal and atomic group multicast. *ACM Transactions on Computer Systems*, 9(3):272–314, August 1991.

[6] K. P. Birman and T. A. Joseph. Exploiting virtual synchrony in distributed systems. In *Proceedings of the 11th ACM Symposium on OS Principles*, pages 123–138, Nov. 1987.

[7] V. Hadzilacos and S. Toueg. Fault-tolerant broadcasts and related problems. In *Distributed Systems*, chapter 5. Addison-Wesley, 2nd edition, 1993.

[8] E. Jul, H. Levy, N. Hutchinson, and A. Black. Fine-grained mobility in the emerald system. *ACM Transactions on Computer Systems*, 6(1), February 1988.

[9] J. Kubiatowicz, D. Bindel, Y. Chen, S. Czerwinski, P. Eaton, D. Geels, R. Gummadi, S. Rhea, H. Weatherspoon, W. Weimer, C. Wells, and B. Zhao. Oceanstore: An architecture for global-scale persistent storage. In *Proc. of the ACM ASPLOS 2000 Technical Conference*. ACM, November 2000.

[10] L. Lamport. Time, clocks, and the ordering of events in a distributed system. *Comm. of the ACM*, 21(7), July 1978.

[11] N. Lynch, D. Malkhi, and D. Ratajczak. Atomic data access in content addressable networks. In *Proc. of the first International Workshop on Peer-to-Peer Systems (IPTPS '02)*, 2002.

[12] P. V. Mockaptris and K. Dunlap. Development of the domain name system. In *Proceedings of the ACM SIGCOMM Technical Conference*. ACM, August 1988.

[13] M. K. Ramanathan, V. Kalogeraki, and J. Pruyne. Finding good peers in peer-to-peer networks. In *Proc. of the International Parallel and Distributed Computing Symposium*, 2002.

[14] S. Ratnasamy, P. Francis, M. Handley, R. Karp, and S. Shenker. A scalable content-addressable network. In *Proceedings of the ACM SIGCOMM 2001 Technical Conference*. ACM, August 2001.

[15] S. Ratnasamy, S. Shenker, and I. Stoica. Routing algorithms for dhts: Some open questions. In *First International Workshop on Peer-to-Peer Systems (IPTPS)*, 2002.

[16] S. C. Rhea and J. Kubiatowicz. Probabilistic location and routing. In *Proceedings of the IEEE INFOCOM 2002 Technical Conference*. IEEE, June 2002.

[17] A. Rowstron and P. Drusche. 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*. ACM, October 2001.

[18] F. B. Schneider. Implementing fault-tolerant services using the state machine approach: A tutorial. *ACM Computing Surveys*, 22(4):299–319, Dec. 1990.

[19] I. Stoica, R. Morris, D. Karger, F. Kaashoek, and H. Balakrishnan. Chord: A scalable peer-to-peer lookup service for internet applications. In *Proceedings of the ACM SIGCOMM 2001 Technical Conference*. ACM, August 2001.

[20] J. R. D. W. J. Bolosky and D. Ely. Evaluation of desktop pcs as candidates for a serverless distributed file system. In *Proceedings of the ACM SIGMETRICS Technical Conference*. ACM, June 2000.