Constructing a Stable Virtual Peer from Multiple Unstable Peers for Fault-tolerant P2P Systems

Masanori Shikano, Kota Abe, Tatsuya Ueda, Hayato Ishibashi and Toshio Matsuura
Graduate School for Creative Cities, Osaka City University
Osaka, Japan
Email: {shikano,k-abe,tueda,ishibashi,matsuura}@sousei.gscc.osaka-cu.ac.jp

***Abstract*—P2P systems must handle unexpected peer failure and leaving, and thus it is more difficult to implement than server-client systems. In this paper, we propose a novel approach to implement P2P systems by using *virtual peers*. A virtual peer consists of multiple unstable peers. A virtual peer is a stable entity; application programs run on a virtual peer are not compromised unless a majority of the peers fail within a short time duration. If a peer in a virtual peer fails, the failed peer is replaced by another (non-failed) one to restore the number of working peers.**

The primary contribution of this paper is to propose a method to form a stable virtual peer over multiple unstable peers. The major challenges are to maintain consistency between multiple peers, to replace a failed peer with another one, and to communicate with a virtual peer, whose member peers are dynamically changed. For the first issue, the Paxos consensus algorithm is used. For the second issue, the process migration technique is used to replicate and transfer a running process to a remote peer. For the last issue, communication protocols based on application level multicast are introduced. Furthermore, the relation between the reliability of a virtual peer and the number of peers assigned to a virtual peer is evaluated.

The proposed method is implemented in our *musasabi* P2P platform. An overview of *musasabi* and its implementation is also given.

***Keywords*—peer-to-peer systems; fault tolerance; Paxos consensus algorithm; process migration; strong mobility**

I. Introduction

In comparison to server-client systems, peer-to-peer (P2P) systems provide superior fault tolerance because P2P systems do not have servers, which are single points of failure. However, because peers in a P2P system are unstable (they fail or leave unexpectedly), P2P systems are more difficult to implement than server-client systems. (In this paper, peer leaving is considered a kind of peer failure and the two are not distinguished.) For example, the following are common measures to achieve fault tolerance in P2P systems: peer failure detection, data replication over multiple peers, and managing multiple pointers as a precaution against peer failure. While such measures are crucial for practical P2P applications to provide stable services, implementing such measures is delicate work and a troublesome burden for developers.

In this paper, we propose a novel approach to ensuring fault tolerance in P2P systems. In the proposed method, *virtual peers* are formed by grouping multiple peers on a P2P network. A virtual peer is an entity for running an application program. Similar to mirroring on file systems, each peer of a virtual peer serves as a part of a redundant system. The running application on a virtual peer is not compromised when some of the peers fail. In addition, when a peer fails, the number of peers is restored by replacing the failed peer with another (non-failed) one. Therefore, virtual peers rarely fail.

Using virtual peers in a P2P system is of great benefit. Developing P2P systems based on virtual peers (i.e., using virtual peers as a building block of P2P systems) is more straightforward than conventional P2P systems; because the possibility of virtual peer failure is negligible, P2P systems constructed over virtual peers do not have to implement measures against peer failures. Furthermore, virtual peers can be used for the following applications.

- Replacing central servers on hybrid P2P systems: One disadvantage of hybrid P2P systems is that central servers are single points of failure. This single point of failure can be eliminated by replacing the server with a virtual peer.
- Using virtual peer as a super peer: In super peer-based P2P systems, super peers have a more important role than normal peers and are expected to be stable. Using a virtual peer as a super peer reduces the possibility of super peer failure and thus the system can be more stable.

The proposed method has been implemented in the prototype of our P2P platform *musasabi*[1]. *Musasabi* is implemented in pure Java.

The rest of this paper is organized as follows. Section II presents the proposed method. Section III provides an overview of *musasabi* and an explanation of the implementation. Section IV describes communication protocols for virtual peers. Section V discusses the proposed method. Section VI gives related work, and Section VII summarizes our conclusion.

II. Proposed method

In the proposed method, multiple peers chosen from the P2P network are grouped to form a *virtual peer* (Figure 1). The peers that form a virtual peer are called *member peers*.

An application program running on a virtual peer is called a *virtual process*. A virtual process corresponds one-to-one with a virtual peer. A virtual process does not stop even if some of the member peers fail. We describe the details below.

[1]*Musasabi* is a type of flying squirrel found in Japan.

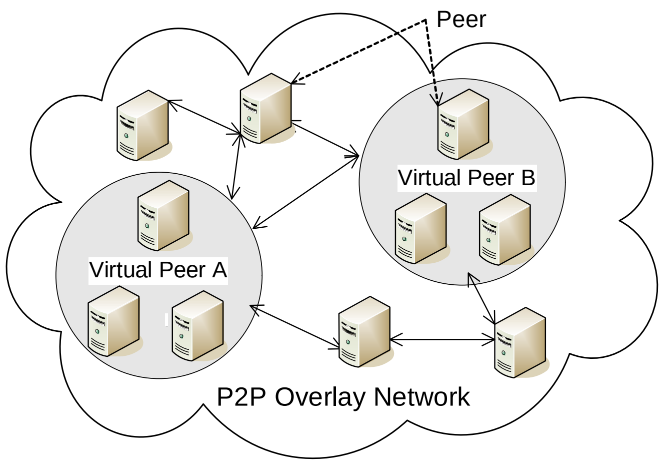

Figure. 1. Peers and Virtual Peers

A. Choosing member peers

In order to start a virtual peer, the initial member peers of the virtual peer must be chosen. Currently, we assume that no nodes in the P2P network decline to be a member peer of any virtual peer. A single peer is allowed to be a member of multiple different virtual peers.

To choose a *good* member peer, the following criteria must be considered: the peer's stability, network distances to the other member peers, network bandwidth, CPU speed and load average, memory size, etc. However, we choose member peers randomly for now.

Choosing a peer randomly can be implemented using a Skip Graph overlay network [3]. We assume that each peer has its own ID (peer ID). On initializing a peer, its peer ID is registered in the Skip Graph, which is shared among all peers. In order to randomly choose a peer, generate a random number r and search for a peer from the Skip Graph whose ID is closest to r.

B. Achieving fault-tolerance of virtual process

To make it possible for a virtual process to continue its execution even if some of the member peers fail, the state of a virtual process must be replicated over multiple member peers. In order to make it easy to develop application programs, we choose the following approach.

Each member peer of a virtual peer simultaneously and redundantly executes the same application program, as a *process*. Each of the processes has the complete replica of the virtual process state. This is depicted in Figure 2.

In order to maintain the state of each process identically, the process must be a state machine. The process state must be changed only by external input (messages) that the process receives. Moreover, the sequences of external input (messages) received by each process are controlled to be completely identical. This approach is known as *State Machine Replication* [4].

In this approach, application programs can be quite simple; just process the received messages in order. No commit protocol is required.

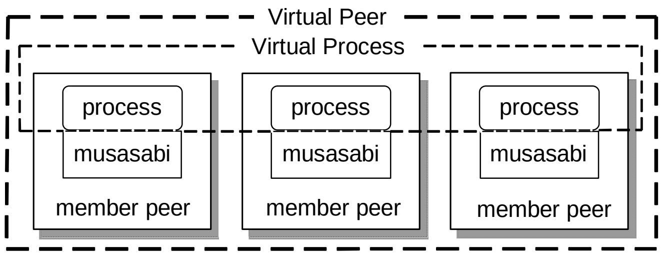

Figure. 2. Virtual peer and virtual process

C. Ensuring identical message sequences

When multiple nodes transmit messages to multiple receiving nodes via the Internet, the message sequences received among the receiving nodes are not necessarily identical. In order to ensure that multiple nodes receive messages in identical order (in other words, *atomic broadcast* or *total order broadcast*), we use the Paxos consensus algorithm because it is proven and well described in literature [4][5].

Paxos is a distributed algorithm to form a consensus between multiple participants (in this case, participants are peers) on an unreliable network. Paxos ensures that all participants (that have not failed) eventually choose a single value that one of the participants proposed. In Paxos, a consensus is reached when a majority of the participants accept the proposed value. Paxos can be extended to a series of values (called Multi-Paxos). In this paper, Paxos and Multi-Paxos are not distinguished.

In order for the Paxos protocol to progress, values must be proposed by only one peer (leader peer). The leader peer is elected from the member peers by a leader election algorithm. Note that the Paxos algorithm guarantees only one value is chosen even if multiple leaders are present. (This is called the *safety* property and is important because leader election may fail to elect a single leader.)

An outline of the Paxos protocol is as follows. The leader peer initially evaluates the current condition by broadcasting *Collect* messages. (Message names are based on the literature [5].) After the leader receives the *Last* message from a majority of the peers, the leader peer can propose a value by broadcasting *Begin* messages. A value is agreed upon when a majority of the peers reply with an *Accept* message. If a value is agreed upon, the leader broadcasts the *Success* message to announce the consensus. Hereafter, consensus about a sequence of values is reached by repeating the Begin–Accept–Success sequence. *Begin*, *Accept* and *Success* messages contain a sequence number to identify each proposal.

As we will describe in Section IV, when a message is sent to a virtual peer, its leader peer receives the message through ALM (Application Level Multicast). In order to make the message order received by each member peer identical, the leader peer assigns a sequential number (i) to the message and proposes the message as the i^{th} value with the Paxos protocol.

On each of the member peers, the agreed-upon message is passed to the process in sequence number order. Note that because the Paxos protocol runs asynchronously, consensus is not always reached in the sequence number order when multiple proposals have been made simultaneously. Therefore, if the sequence number of a received message is not continuous with the previous one (i.e., if there is a *gap* between them), the process must wait until the gap is filled. If the gap is not filled for a long time, a retransmission request is sent to the leader peer.

D. Handling peer failure

If a member peer fails, the remaining member peers must replace the failed peer with another one to keep the number of member peers constant. Otherwise, eventually the virtual peer will *evaporate*.

In order to detect member peer failure, each member peer carries out watchdog monitoring on the others. When a peer detects that another peer has failed, the failed peer is replaced by another one. As mentioned in Section II-A, a substitute peer is randomly chosen from the peers in the P2P network.

All of the member peers must maintain a consistent view of the member configuration. Paxos is also used to change the peer configuration in order to ensure consistent updating of this view. The procedure is as follows.

i) If the failed peer is the leader, a new leader is elected (as explained later in Section III-D).
ii) The leader peer chooses another peer p from the P2P network.
iii) The leader peer proposes a peer configuration change (swapping the failed peer with p) using Paxos.
iv) When the proposal has been agreed upon, the peer configuration is changed by all of the member peers. If the consensus fails, return to 2) after waiting briefly.
v) p must execute the same process, whose state must be identical to the ones on other member peers. For this reason, the *process migration technique* is used to replicate the running process on the leader peer to p (see Section III-A).

In order to reach consensus in Paxos, a majority of the peers must be functional while replacing a failed peer. Otherwise, the virtual peer fails. Reliability of virtual peers is discussed in Section V-A.

Note that some of the member peers might not be aware that the configuration has been changed, partly because the *Success* message is lost, and partly because only a majority of member peers are required to reach a consensus. If such a peer becomes the leader, consensus might not be reached because it may not be able to reach a majority of member peers. To avoid this situation, member configuration is periodically exchanged among the member peers. The details of this issue is out of the scope of this paper. We will describe this in another article.

E. Communication with a virtual peer

Communication protocols for virtual peers are separately discussed in Section IV because they require much space.

III. P2P PLATFORM *musasabi*

We implemented the proposed method in the prototype of our *musasabi* P2P platform. *Musasabi* is written in Java and is intended to be a platform for implementing P2P services. Each instance of *musasabi* can be regarded as a peer.

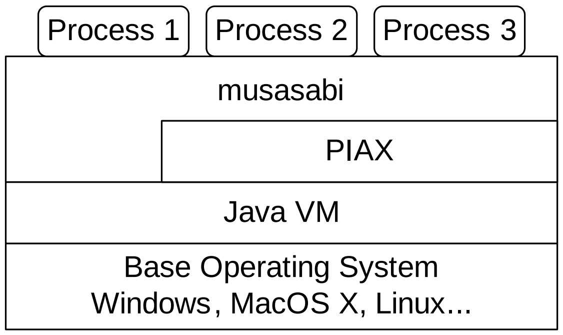

Figure. 3. Configuration of *musasabi*

On *musasabi*, an application program, also in Java, can be executed as a *process*. To implement a P2P service using *musasabi*, a process on *musasabi* communicates with other processes on remote *musasabi* peers.

Because a process may be transferred from a remote peer using process migration, measures against malicious programs are necessary in order to protect a local node. For this reason, processes are executed in the Java sandbox mechanism.

Musasabi uses another P2P platform, *PIAX* [6], for P2P networking. Among the functions provided by PIAX, the Skip Graph overlay network and the ALM service are important for *musasabi*. The configuration of *musasabi* is shown in Figure 3.

Musasabi supports the virtual peer mechanism described above; we describe the details below.

A. Implementation of process migration

In order to implement a virtual peer, *process migration*, a function to transfer a running process from a node to another node, is required.

Process migration techniques are classified into two classes: *weak mobility* and *strong mobility* [7]. In weak mobility, only program code and data fields (the values of global variables) are transferred, whereas in strong mobility, in addition to these, the execution context of threads (the contents of the thread stack and the value of its program counter) is also transferred. Strong mobility makes it possible to describe mobile programs in a natural form. *Musasabi* supports strong mobility, whereas the standard Java does not.

Musasabi provides the following APIs for process migration. An example of their use is shown in Figure 4.

go(peer)
: The calling process is migrated to the specified peer.

fork(peer)
: The calling process is replicated and the replicated process is transferred to the specified peer. This API resembles the *fork* system call in UNIX.

1) Implementation of strong mobility: Some research has been done on implementing strong mobility in Java. As stated above, in order to achieve strong mobility, program code, data fields, and execution contexts must be transferred. The program code can be transferred as class (or jar) files. The Java

```
// start on peer p0
PeerId p1 = ...;
go(p1); // move to peer p1
...; // run on p1
PeerId p2 = ...;
// duplicate the process and transfer to peer p2
if (fork(p2) == null) {
  ...; // run on p2
} else {
  ...; // run on p1
}
```

Figure. 4. A sample program to demonstrate process migration APIs of *musasabi*

```
class MyClass() implements Runnable {
  public void run() {
    block1;
    Continuation.suspend(); // suspend here
    block2;
  }
};
// start from MyClass#run().
// it will be suspended after executing block1 and
// the executing state is saved to c.
Continuation c
  = Continuation.startWith(new MyClass());
// resume from block2.
c = Continuation.continueWith(c);
```

Figure. 5. A sample program of Javaflow's Continuation

serialization mechanism can be used to transfer the data fields. The remaining problem is the method for transferring the execution context. The following methods are known [8]: (1) modifying the Java Virtual Machine (JVM) [9], (2) extending JVM using the Java Native Interface that is provided by JVM [10], (3) translating Java byte code [8][11], and (4) translating Java source code [12].

In a P2P network, because each peer is independently managed, it is difficult to use a non-standard JVM or native code. Thus, (1) and (2) are rejected. *Musasabi* adopts method (3), using *Javaflow*, a library for realizing *Continuation* in Java, because it is publicly available [13]. The idea of using Javaflow for implementing strong mobility is also described in the literature [8].

Javaflow allows Java programs to save a program execution context as an object called *continuation*, which contains information of the stack frame of a thread. Saved context can be restarted later (Figure 5). To make it possible to save and restore an execution context, Javaflow translates the byte code of the program based on certain rules. Both Javaflow and translated byte codes run on a standard JVM.

Because continuation of Javaflow is *Serializable*, it is possible to restart the saved context on a different node. Thus, strong mobility can be implemented using Javaflow. (Note that to serialize a continuation, all objects stored in the saved context also must be *Serializable*.)

2) Migration of a multi-threaded process: When a multi-threaded process is migrated by using the strong mobility model, the execution contexts for all of the threads must be stored and recovered. However, by using a method in which JVM is not modified (including the case of using Javaflow), the execution context cannot be forcibly acquired from outside of the thread; in order to acquire the execution context of a thread, the thread itself must call an API to capture the context. Therefore, it is difficult to migrate all the threads in a process when a thread requests a process migration.

Musasabi solves this problem by providing *Coroutines* instead of Java threads. Coroutines are a unit of parallel processing, similar to threads. While threads may be executed simultaneously (in parallel), coroutines are executed in turn. While context switching between multiple threads is preemptive, context switching between coroutines is voluntary. When a coroutine is running, other coroutines are not executed until the running coroutine itself yields the execution right (by executing yield() or some other API). Therefore, when a coroutine requests process migration, all other coroutines should have yielded, and thus the execution state of all the coroutines can be transferred to the destination peer. The APIs for coroutines in *musasabi* are similar to the standard Java thread APIs.

Javaflow is also used to implement coroutines in *musasabi*. Context switching between coroutines is done by saving a coroutine context and restoring another coroutine context. A simple coroutine scheduler is also implemented.

B. Application Level Multicast

As we will describe in Section IV, the proposed method uses ALM for sending messages to a virtual peer. *Musasabi* uses the ALM service of PIAX, which is built on top of Skip Graph. The implementation of ALM is simple. If a peer joins a multicast group whose ID is g, the peer registers g as a key in the Skip Graph. A message sent to the multicast group g is routed over the Skip Graph and distributed to all peers that registered the key g.

Note that communication within a virtual peer is directly performed on the IP layer.

C. Starting a virtual peer

A virtual peer (and the corresponding virtual process) is started as follows. First, a user starts a normal (non-virtual) process on a peer on which *musasabi* is running. This peer will be the initial leader. The process requests *musasabi* to become a virtual process. An ID for the virtual peer is assigned using a random number generator. Then, in order to form a virtual peer, *musasabi* randomly chooses the specified number of member peers from the P2P network. The initial leader peer joins the multicast group whose ID is the same as the ID of the virtual peer.

D. Leader election

If a leader peer fails, the Paxos protocol needs a new leader. In *musasabi*, a new leader is elected as follows.

Failure of a member peer is detected by timeout of the *Keep Alive* message, which each of the member peers periodically sends to others. Each member peer maintains the (*Alive* or *Dead*) status of the other member peers.

Each member peer has a unique number, *mnum* (member number). All member peers know the *mnum* of the others. If the number of the member peers is n, the *mnum* of each initial n peer will be $0, 1, \ldots, n-1$. When a peer replacement is agreed upon, the *mnum* of the new peer must be assigned without duplication. In order to guarantee the uniqueness of *mnum* even in a multiple leader situation, we use the Paxos sequence number which is used in the replacement proposal as the new *mnum*. (The initial Paxos sequence number is adjusted not to be duplicated with the initial *mnum*s.)

If a leader peer fails, the next leader is the member peer that is alive and whose *mnum* is the smallest. When a member peer detects that the leader peer has failed, the peer determines whether the peer should be the next leader (check whether the peer itself has the smallest *mnum* among all live member peers). If the peer believes that it is the next leader, the peer broadcasts *Collect* messages (Section II-C). If the peer receives *Last* messages from a majority of the peers, it joins in the multicast group of the virtual peer and broadcasts *IamLeader* messages to announce that it is the new leader. If the peer fails to collect enough *Last* messages (this could happen if another peer also believes that it should be the next leader), and no *IamLeader* message has been received from other member peers, the peer retries broadcasting *Collect* messages after waiting for some random period.

E. Replacing a failed peer

When a configuration change proposal (swap the failed peer with a substitute peer p) is agreed-upon, the leader peer uses the fork() API to replicate the process running on the leader peer onto peer p.

An outline of a replacement sequence for a failed peer is shown in Figure 6. The leader peer detects that Peer#2 has failed because of a *Keep Alive* timeout, and chooses Peer#3 as a substitute. The leader proposes replacing Peer#2 with Peer#3. When the proposal is agreed upon, the member configuration is changed in each member peer. In addition, the leader peer replicates (fork) the process onto Peer#3. Subsequently, the virtual peer (process) is served by peers #0, #1, and #3.

F. Interaction between application programs and Paxos

Interaction between application programs and Paxos is depicted in Figure 7. Messages that the leader peer receives via ALM are not automatically proposed by *musasabi*. All messages received via ALM are sent to the process on the leader to determine whether the message should be proposed or not. This is because messages that do not affect the process' status, such as simple read requests, may be processed without proposing. (However, not all read requests can be processed without proposing; if multiple leaders are present, a leader cannot guarantee that it has the latest state. Therefore, only messages that do not depend on the latest state can be processed without proposing.)

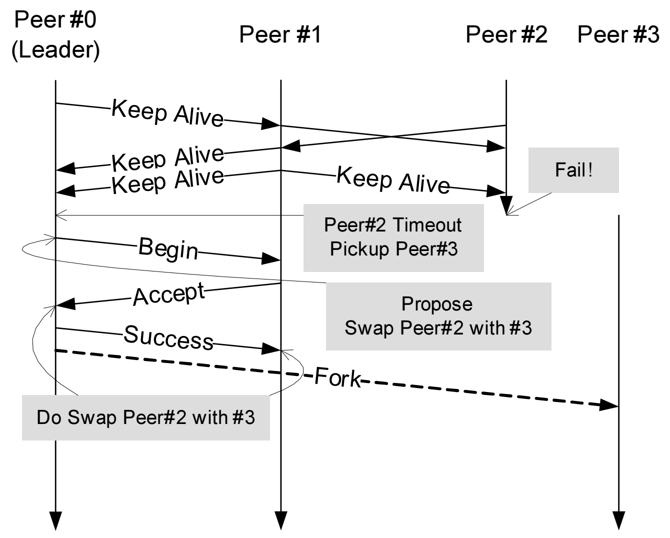

Figure. 6. Sequence for replacing a failed peer

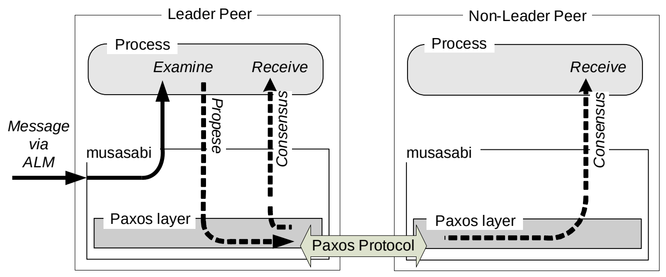

Figure. 7. Interaction between Applications and Paxos

IV. Communication protocols for virtual peers

To implement services using virtual peers, virtual peers must be able to communicate with other peers (either virtual or non-virtual). Communication between virtual peers is a kind of group communication, which is a means for multi-point to multi-point communication. In this section, we describe communication protocols for virtual peers.

To clearly distinguish the two, we call a peer that is not a virtual peer a **normal peer**. We assume that a peer (either normal or virtual) sends a request message to a virtual peer and receives a response message. We also assume that messages in the underlying network may be delayed or lost. In our protocols, communication channels between a sender and a receiver are not FIFO, i.e., if multiple messages are asynchronously sent, their order at the receiver may be changed.

The structure of our protocols is depicted in Figure 8. Note that basically each application in the figure is executed on a different node.

We describe the communication protocol between a normal peer and a virtual peer in Section IV-A, and between virtual peers in Section IV-B.

A. Communication between a normal peer and a virtual peer

First, we discuss the protocol in the case where a normal peer p sends a message m to a virtual peer v and v returns a response message q to p.

1) Delivery message to a virtual peer: As described in Section II-D, member peers of a virtual peer are not fixed. Thus, it is not a straightforward task to deliver messages to a virtual peer from outside of the virtual peer.

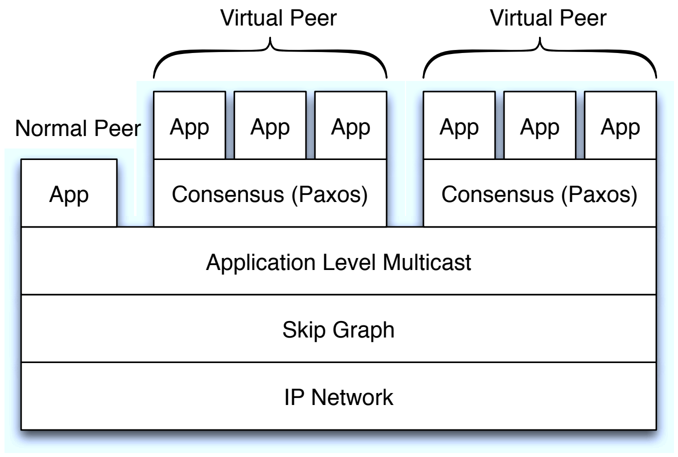

Figure. 8. Protocol Structure

This issue can be solved using ALM. All peers of a virtual peer join in a multicast group specified by the ID of the virtual peer. When a peer sends a message to a virtual peer, the peer multicasts the message over the multicast group of the virtual peer. The sender peer does not have to know the addresses of each member peer. (Note that multicast is required because there might be multiple leaders in a single virtual peer (See Section II-C).)

2) Retransmission of message: When the leader peer of v receives m through ALM, it proposes m with the Paxos protocol, as described in Section II-C. (All the other peers of v do not propose m). Each process of each member peer receives and processes the agreed-upon message in sequence number order. However, proposed messages might not be agreed upon because, for example, the message might be lost, or even if the message arrives at the leader, the leader might leave before proposing the message. For this reason, p has a response timer and periodically retransmits the message m until p receives q.

3) Message ID: As messages might be retransmitted, a receiver may receive duplicated messages. In order to detect such duplicated messages, a message ID is assigned to each message. To uniquely assign a message ID to a message, message IDs are generated by combining the peer ID of p and the sequence number of the message. Message IDs are also used on sender peers for finding a request message from a received response message.

4) Sending response: On receiving m on v, each process on v must call an API to send a response message q to p. Note that all the responses that each process produces must be identical in our execution model (see Section II-B). To reduce network traffic, only the leader peer actually sends the response.

5) Retransmission of response: Response messages from the leader peer also might be lost. In such a case, p retransmits m to v when the response timer expires. However, since m has already been processed by v in this case, v should not reprocess the retransmitted message and just resend the response message previously sent. For this reason, v keeps response messages that it has sent for a while.

6) Protocol: We propose the following protocol based on the discussion above.

Behavior of sender: The peer p generates a message ID for m and multicasts m to the multicast group of v. The peer p retransmits m if no response message for m is received within a fixed period of time.

Behavior of a receiver: The algorithm for a receiver virtual peer is implemented on each member peer.

On each member peer, a *history* table, which keeps the status of received messages, and a *response* table, which keeps response messages, are maintained. These are used for preventing the processing of the same message more than once. These tables can be implemented as hash tables whose key is a message ID. The entry of the *history* table is one of the following:

INITIAL
The message has not yet been received (default).

PASSED
The message has been passed on for processing.

REPLIED
The message has been passed on for processing and its response message has been sent.

The algorithm of a receiver is shown below.

i) On receiving m through ALM, only the leader peer executes the following steps. (all other peers just ignore m).
   a) Get the *status* of m from the *history* table.
   b) If the *status* is INITIAL, then propose m using Paxos.
   c) If the *status* is PASSED, then just ignore m.
   d) If the *status* is REPLIED, then send the response message recorded in the *response* table to p.

ii) On receiving the *Success* message of m (notification of agreement), all the member peers execute the following steps.
   a) Get the *status* of m from the *history* table.
   b) If the *status* is INITIAL, then (1) record the status of m as PASSED in the *history* table, and (2) pass m to the process on the peer.
   c) If the *status* is not INITIAL, then just ignore m.

iii) The process that received m must call an API to send a response message q to p. In the API, following steps are executed.
   a) Record the status of m as REPLIED in the *history* table.
   b) Record q in the *response* table.
   c) Only the leader peer sends q.

7) Examples of the message sequence: An example of message sequence in which no message retransmission occurs is shown in Figure 9, and another example in which message retransmission occurs is shown in Figure 10. The bold short dashed lines in the figures mean the period the process is handling the message.

B. Communication between virtual peers

Next, we discuss the protocol in the case where a virtual peer s sends a message m to another virtual peer r and r

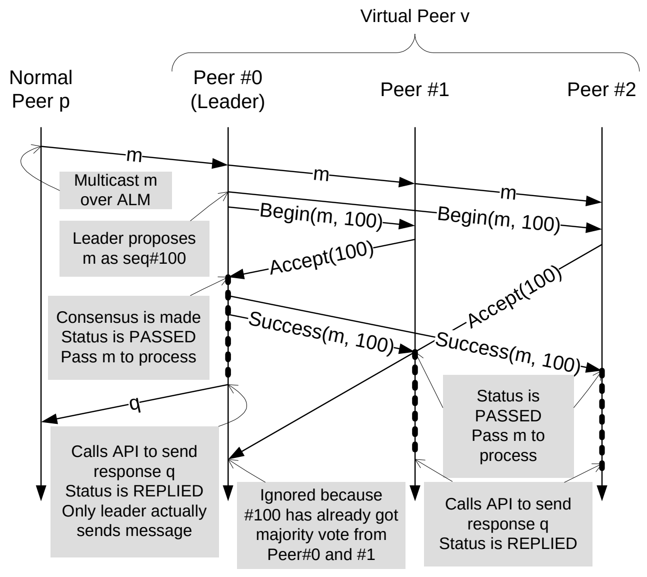
Virtual Peer v
Normal Peer p
Peer #0 (Leader)
Peer #1
Peer #2
m
Multicast m over ALM
Begin(m, 100)
Leader proposes m as seq#100
Accept(100)
Consensus is made Status is PASSED Pass m to process
Success(m, 100)
q
Status is PASSED Pass m to process
Calls API to send response q Status is REPLIED Only leader actually sends message
Ignored because #100 has already got majority vote from Peer#0 and #1
Calls API to send response q Status is REPLIED

Figure. 9. Example of message sequence between a normal peer and a virtual peer (without retransmission case)

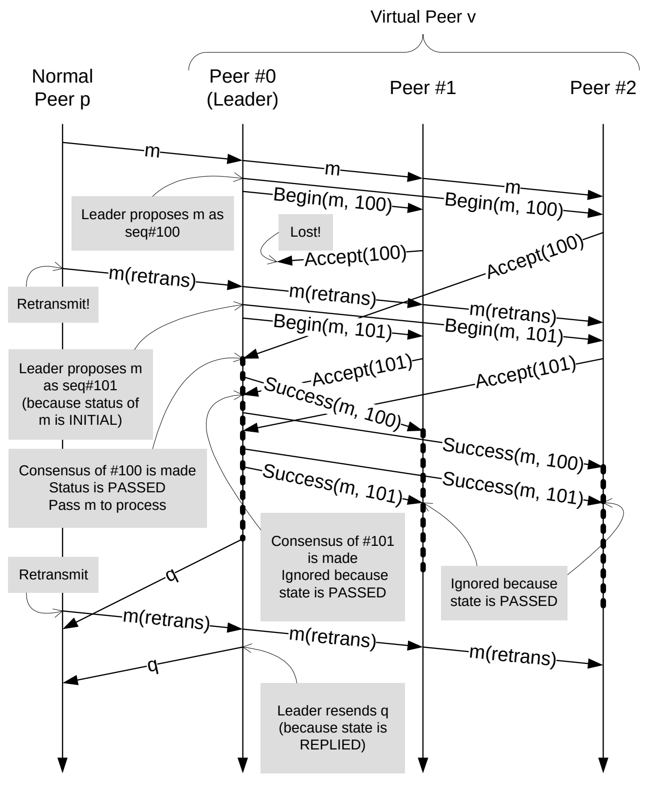
Virtual Peer v
Normal Peer p
Peer #0 (Leader)
Peer #1
Peer #2
m
Begin(m, 100)
Leader proposes m as seq#100
Lost!
Accept(100)
m(retrans)
Retransmit!
Begin(m, 101)
Leader proposes m as seq#101 (because status of m is INITIAL)
Accept(101)
Success(m, 100)
Consensus of #100 is made Status is PASSED Pass m to process
Success(m, 101)
Consensus of #101 is made Ignored because state is PASSED
Ignored because state is PASSED
Retransmit
q
Leader resends q (because state is REPLIED)

Figure. 10. Example of message sequence between a normal peer and a virtual peer (with retransmission case)

returns a response message q to s. Note that if s sends m to r, s must have received a *trigger* message M, which has been agreed upon, because virtual processes are executed in message-driven models. We assume that each process of s calls an API to send m to r by receiving M.

Communication between virtual peers is basically the same as that between a normal peer and a virtual peer. A leader peer of s plays the role of a sender peer p in Section IV-A. In addition to that, the following points should be considered.

1) Sending response: A response message from a virtual peer r also must be sent with ALM because member peers of s may be changed while s is communicating with r.

2) Agreement of response: Response messages also must be ordered identically at all of the member peers because message receiving order may affect the internal state of a virtual process. Therefore, the leader of s must propose every received response message with the Paxos protocol. A response message is processed after it is agreed upon.

3) Member peers change: Here we consider a case where the leader peer s_l of s leaves before receiving a response message from r, after M is agreed upon. Suppose the following scenarios.

- s_l leaves before sending m to r: this includes the two cases shown below.
  - (Case 1a) The next leader s_l' of s has processed M (i.e., M has been passed to the process on s_l') before it becomes a leader.
    In this case, when s_l' called the API to send m to r, no messages were actually sent by s_l' because it was not a leader at that time. However, when the response timer expires, s_l' must actually send m this time. For this reason, all the peers should record their sending messages whether they are leaders or not.
  - (Case 1b) s_l' processes M after it becomes a leader. In this case, m is sent to r by s_l' when it processes M.
- s_l leaves after m is sent to r: it is possible to classify it into two groups as follows.
  - (Case 2a) s_l' has been elected as a leader when q arrives from r:
    In this case, s_l' simply proposes q with Paxos.
  - (Case 2b) s_l has already left but no leader has been elected when q arrives:
    In this case, because no peer in s proposes q and thus q is lost, the next leader (s_l') retransmits m to get q as soon as it is elected.

4) Protocol: We propose the following protocol based on the discussion above.

Behavior of sender: The algorithm for a sender virtual peer is implemented on each member peer.

i) In the API to send m to r, the following steps are executed. (Note that all the processes in s call this API on receiving M.)
   a) In our execution model, all member peers must independently generate the same message m. Therefore, the message ID of m also must be identical. Thus, message IDs are generated based on not the ID of each member peer but the ID of virtual peer s.
   b) Only the leader peer multicasts m to the multicast group of s.
   c) Record m for retransmitting and start the response timer.
   d) When the response timer expires, only the leader peer retransmits m.

ii) On receiving q through ALM, s processes q according to the receiver protocol i) and ii) described in Section IV-A6. All the member peers of s stop the timer after step ii).

If the leader peer that sent m leaves and the new leader detects that no response message for m has been received, the new leader retransmits m, as described in Section IV-B3.

Behavior of receiver: The behavior of receiver r is almost the same as that of receiver v described in Section IV-A6, except that the leader peer of r multicasts the response message to the multicast group of s.

5) Example of message sequence: An example of a message sequence between virtual peers is shown in Figure 11.

V. Discussion

A. Reliability of virtual peers

In this section, we discuss the relation between the reliability of a virtual peer and the number of its member peers.

Let t be elapsed time since the start of a virtual peer, m the number of member peers, and T the maximum time required to swap a failed peer (from the moment of a peer failure until fork() is done). Because the Paxos protocol requires that a majority of member peers survive to reach a consensus, and because Paxos is used for changing a peer configuration, a virtual peer looses its functionality when $\lfloor (m+1)/2 \rfloor$ (denoted as n) peers have failed during T. Assuming that each peer fails independently and that the intervals of peer failure are exponentially distributed, let λ' be the peer failure rate per unit of time. λ' can be expressed using the duration of half of the peers failing (denoted as t_{half}), as $e^{\lambda' t_{\text{half}}} = 0.5$. The reliability of a single peer $R'(t)$ is expressed as $R'(t) = e^{-\lambda' t}$. Because the probability that n peers fail within time duration T is $(1-R'(T))^n$, the failure rate of a virtual peer λ is expressed as $\lambda = (1 - R'(T))^n/T$. Note that the reliability of a virtual peer $R(t)$ is given by $R(t) = e^{-\lambda t}$.

The value of T mainly depends on the *Keep Alive* message interval. We pessimistically assume that $T = 60$ (sec).

Now, we show two graphs (Figure 12 and Figure 13). Figure 12 shows the reliability of virtual peers for a different number of member peers versus t. Because the peer failure rate depends on the environment, here we assume $t_{\text{half}} = 1$ (hour). In this case, a virtual peer is practically stable if it consists of seven member peers.

Next, we vary the peer failure rate and show the resulting MTTF (Mean Time To Failure) of a virtual peer (expressed as λ^{-1}) in Figure 13. Note that the x-axis is expressed in t_{half}. The graph indicates that, even in the high peer failure rate environment, virtual peers can be practically stable by increasing the number of member peers. (We modestly assume that in the real environment $t_{\text{half}} > 10$ (min) because only 1.56% of peers remain after one hour if $t_{\text{half}} = 10$, which seems too pessimistic.)

B. Overhead of the proposed method

In this section, we discuss the overhead of the proposed method.

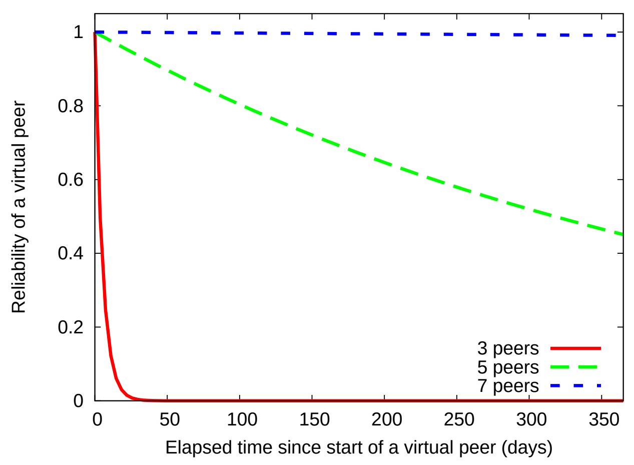

Figure. 12. Reliability of a virtual peer.

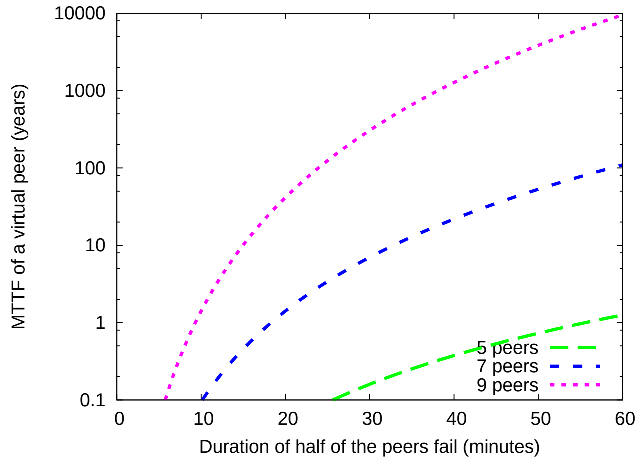

Figure. 13. MTTF of a virtual peer versus peer failure rate.

1) Overhead of the Paxos: The Paxos protocol imposes latency on processing a request sent to a virtual peer. The latency is mostly affected by the RTT (Round Trip Time) between the leader and other member peers. (Note that the latency is not proportionally increased with the number of member peers because a leader peer can send *Begin* messages to each member peer without waiting.)

2) Overhead of communication: The ALM, used for sending a message to a virtual peer, also imposes extra overhead. Because *musasabi* uses ALM built on Skip Graph, sending a message over ALM requires $O(\log n)$ hops where n is the number of virtual peers. This overhead is common in P2P systems but can be reduced if a peer subsequently communicates with the same virtual peer. A peer first uses ALM to send a message and receives the configuration of the virtual peer; subsequent messages can be sent directly to the leader peer (until it fails).

3) Overhead of watchdog monitoring: If one wants to make the maximum time required to swap a failed peer below 60 seconds (see Section V-A), the period for sending *Keep Alive* messages will be around 20 to 30 seconds. This overhead is small and will not be a problem, we believe.

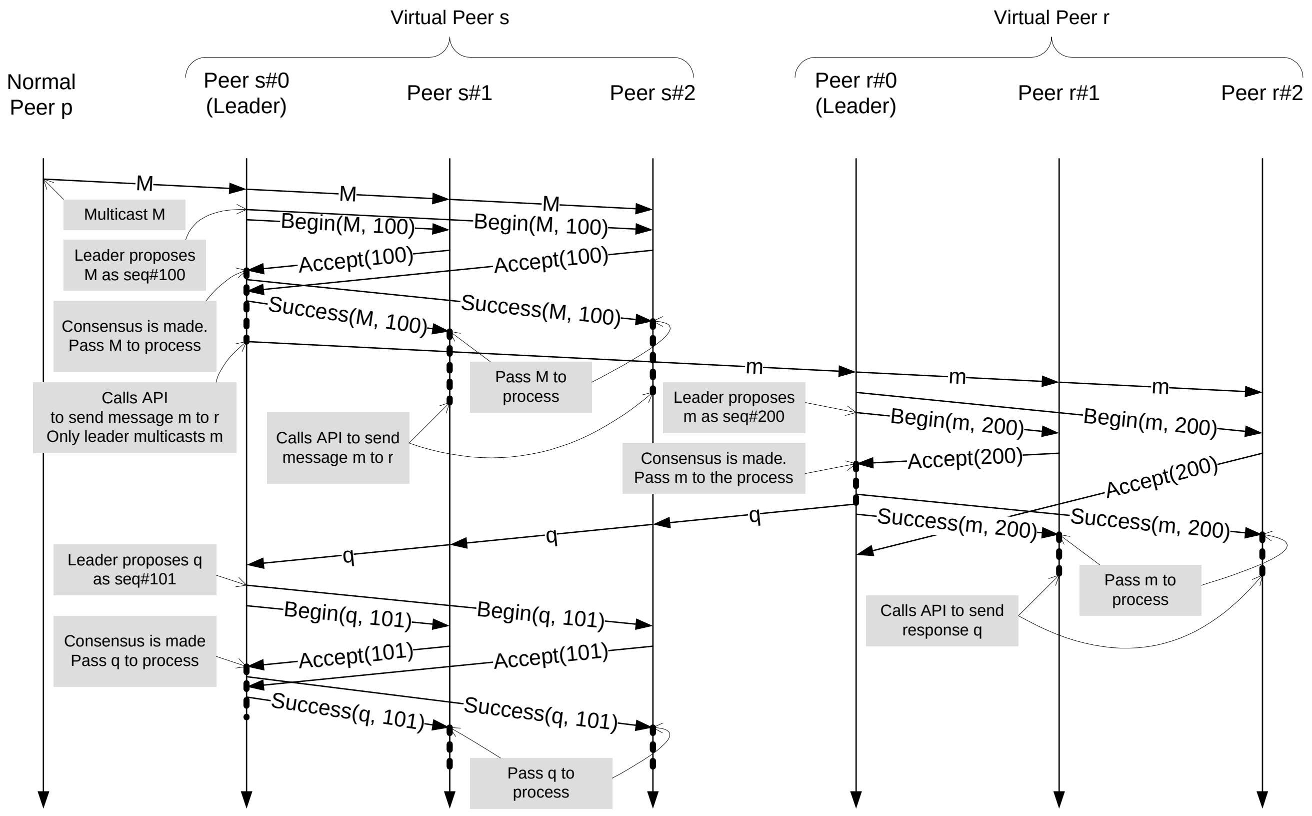
Virtual Peer s
Virtual Peer r
Normal Peer p
Peer s#0 (Leader)
Peer s#1
Peer s#2
Peer r#0 (Leader)
Peer r#1
Peer r#2
M
Multicast M
Begin(M, 100)
Leader proposes M as seq#100
Accept(100)
Success(M, 100)
Consensus is made. Pass M to process
Pass M to process
Calls API to send message m to r Only leader multicasts m
Calls API to send message m to r
m
Leader proposes m as seq#200
Begin(m, 200)
Consensus is made. Pass m to the process
Accept(200)
Success(m, 200)
q
Pass m to process
Calls API to send response q
Leader proposes q as seq#101
Begin(q, 101)
Consensus is made Pass q to process
Accept(101)
Success(q, 101)
Pass q to process

Figure. 11. Example of message sequence between virtual peers

4) Coroutines and byte code translation: The performance of coroutines is inferior to that of threads, especially on multi-CPU machines. In addition, the byte code translation technique used for achieving strong mobility may degrade the performance of application programs. We believe these will not be a problem unless virtual peers are used for computational purposes (such as in P2P-Grid).

C. Leader failure

If the leader peer fails, execution of the virtual process stops until the new leader is elected. Therefore, leader failure must be quickly detected. Multi-coordinated Paxos [14], which allows multiple leaders and thus improves the availability of the system, may relax this problem.

D. Network partitioning

If network partitioning occurs and no fragment of the network contains a majority of the member peers, the virtual peer looses its functionality. (We assume an odd number of member peers.) In this case, the virtual peer must wait until the network has recovered. Note that this situation happens only if the network is split into three or more fragments.

E. Scalability

Using a virtual peer does not contribute to scalability problems of P2P systems. Scalability can be obtained by using multiple virtual peers.

F. Security

A virtual peer might be compromised either if a malicious node is selected as the leader peer or if a member peer violates the Paxos protocol. The Byzantine Paxos protocol [15] may ameliorate this issue but this kind of protocol imposes big overhead. This is beyond the scope of this paper and will be investigated in the future.

VI. Related work

There are few research efforts that use the Paxos algorithm in P2P-related contexts. *Scalaris*, a distributed transactional key/value store written in *Erlang*, uses an adapted Paxos protocol to implement replication and ACID properties (atomicity, concurrency, isolation, durability) [16]. In the field of cloud computing, Google's *Chubby*, an implementation of distributed lock service, uses Paxos in order to make a fault-tolerant distributed database with multiple computers [17][18]. However, these systems do not provide fault-tolerance of general applications.

In the field of Grid computing, *Vigne*, a grid middleware for dynamic large scale grids, uses similar techniques (such as execution on redundant nodes, state machine replication using atomic multicasting over the Paxos protocol) to achieve fault-tolerance of its *Application Manager* [19]. The distinguished differences are that (1) *Vigne* uses the Pastry overlay network [20] for routing messages to the manager node (which corresponds a leader peer in *musasabi*), whereas *musasabi* uses ALM over Skip Graph, and that (2) replicas of the manager node are selected from the leaf set (peers whose ID's are close) of Pastry. The drawbacks of this approach are that there is less flexibility on choosing replica members and that it incurs reconfiguration of replica members not only in case of node failure but also in case of node addition (in the leaf set) [19].

In the literature [21], Mena et al. proposes a group communication scheme based on consensus. While their protocol stack is similar to ours, their discussion is generic and not specific for P2P network. The contribution of our paper is that we revealed the detail of how to implement such architecture in P2P network.

VII. Conclusion and Future Work

P2P systems must handle unexpected peer failure and leaving, and thus they are more difficult to implement than server-client systems.

In this paper, we proposed a method to construct a stable virtual peer from multiple unstable peers. An application program running on virtual peers is not compromised unless a majority of the underlying unstable peers fail within a short time duration. Moreover, application programs of this method can be quite simple. Virtual peers achieve superior fault-tolerance by integrating the Paxos consensus algorithm and process migration technique. In addition, we proposed communication protocols for virtual peers based on application level multicast, and analyzed the relation between the reliability of a virtual peer and the number of peers assigned for a virtual peer. The result indicates that our method appears promising.

The proposed method can be used for reducing development costs, and for improving stability, of P2P systems.

The proposed method has some overhead as described in Section V-B. Quantitative evaluation of each overhead is one of our future work. Other future work includes: (1) improving the method for choosing good member peers, (2) analyzing and improving security of virtual peers, and (3) evaluating the method on the Internet.

Acknowledgment

We would like to thank Dr. Mikio Yoshida, the developer of PIAX at BBR Inc., for giving helpful advice regarding PIAX. This research was partially supported by National Institute of Information and Communications Technology (NICT), Japan.

References

[1] Kota Abe, Tatsuya Ueda, Masanori Shikano, Hayato Ishibashi, and Toshio Matsuura. Toward Fault-tolerant P2P Systems: Constructing a Stable Virtual Peer from Multiple Unstable Peers. In *AP2PS '09: Proc. of 1st Intl. Conf. on Advances in P2P Systems*, pages 104–110. Information Processing Society of Japan, 2009.

[2] Masanori Shikano, Tatsuya Ueda, Kota Abe, Hayato Ishibashi, and Toshio Matsuura. Communication Methods for Virtual Peers on musasabi P2P Platform. Technical Report 2, IPSJ DPS-139, 2009 (in Japanese).

[3] James Aspnes and Gauri Shah. Skip graphs. *ACM Trans. on Algorithms*, 3(4):1–25, 2007.

[4] Leslie Lamport and Keith Marzullo. The part-time parliament. *ACM Trans. on Computer Systems*, 16:133–169, 1998.

[5] Roberto De Prisco and Butler Lampson. Revisiting the Paxos algorithm. In *Proc. of 11th Intl. Workshop on Distributed Algorithms (WDAG 97)*, pages 111–125. Springer-Verlag, 1997.

[6] Mikio Yoshida, Takeshi Okuda, Yuuichi Teranishi, Kaname Harumoto, and Shinji Shimojo. PIAX: A P2P Platform for Integration of Multi-overlay and Distributed Agent Mechanisms. *IPSJ Journal*, 49(1):402–413, 2008 (in Japanese).

[7] Alfonso Fuggetta, Gian Pietro Picco, and Giovanni Vigna. Understanding code mobility. *IEEE Trans. on Software Engineering*, 24:342–361, 1998.

[8] Jose Ortega-Ruiz, Torsen Curdt, and Joan Ametller-Esquerra. Continuation-based mobile agent migration. ⟨http://hacks-galore.org/jao/spasm.pdf⟩. (23-Jan-2009).

[9] Anurag Acharya, M. Ranganathan, and Joel Saltz. Sumatra: A language for resource-aware mobile programs. In *Mobile Object Systems Towards the Programmable Internet*, pages 111–130. Springer-Verlag, 1997.

[10] Wenzhang Zhu, Cho li Wang, and Francis C. M. Lau. JESSICA2: A distributed Java virtual machine with transparent thread migration support. In *IEEE 4th Intl. Conf. on Cluster Computing*, 2002.

[11] Takahiro Sakamoto, Tatsurou Sekiguchi, and Akinori Yonezawa. Bytecode transformation for portable thread migration in Java. In *Joint Symposium on Agent Systems and Applications/Mobile Agents*, pages 16–28, 2000.

[12] Arjav J. Chakravarti, Xiaojin Wang, Jason O. Hallstrom, and Gerald Baumgartner. Implementation of strong mobility for multi-threaded agents in Java. In *Proc. of Intl. Conf. on Parallel Processing*, pages 321–330. IEEE Computer Society, 2003.

[13] Apache Project. Javaflow. ⟨http://commons.apache.org/sandbox/javaflow/⟩. (23-Jan-2009).

[14] Lásaro Jonas Camargos, Rodrigo Malta Schmidt, and Fernando Pedone. Multicoordinated paxos. In *PODC'07: Proc. of the 26th annual ACM sympo. on Principles of distributed computing*, pages 316–317. ACM, 2007.

[15] Miguel Castro and Barbara Liskov. Practical byzantine fault tolerance. In *3rd Sympo. on Operating Systems Design and Implementation (OSDI)*. USENIX Assoc., Co-sponsored by IEEE TCOS and ACM SIGOPS, 1999.

[16] Thorsten Schütt, Florian Schintke, and Alexander Reinefeld. Scalaris: reliable transactional p2p key/value store. In *ERLANG '08: Proc. of 7th ACM SIGPLAN workshop on ERLANG*, pages 41–48. ACM, 2008.

[17] Mike Burrows. The chubby lock service for loosely-coupled distributed systems. In *OSDI '06: Proc. of 7th sympo. on Operating systems design and implementation*, pages 335–350. USENIX Assoc., 2006.

[18] Tushar D. Chandra, Robert Griesemer, and Joshua Redstone. Paxos made live: an engineering perspective. In *PODC'07: Proc. of 26th annual ACM symposium on Principles of distributed computing*, pages 398–407. ACM Press, 2007.

[19] R. K. Nath. Fault tolerance of the application manager in Vigne. Master's thesis, University of Tennessee, 2008. Internship Report.

[20] Antony I. T. Rowstron and Peter Druschel. Pastry: Scalable, decentralized object location and routing for large-scale peer-to-peer systems. *IFIP/ACM Intl. Conf. on Distributed Systems Platforms (Middleware)*, 2218:329–350, 2001.

[21] Sergio Mena, André Schiper, and Pawel Wojciechowski. A step towards a new generation of group communication systems. In *Proc. of the ACM/IFIP/USENIX 2003 Intl. Conf. on Middleware*, pages 414–432. Springer-Verlag, 2003.