Fast Recovery in Distributed Shared Virtual Memory Systems

Va-On Tam  Meichun Hsu
Aiken Computation Lab.,
Harvard University
Cambridge, MA 02138
tam@harvard.harvard.edu
hsu@harvard.harvard.edu

Abstract:

Distributed shared virtual memory (DSVM) is an abstraction which integrates the local memory of different machines in a local area network environment into a single logical entity. In this paper we study the problem of system failure and recovery of DSVM. Most DSVM systems use the notion of tokens to indicate a site's access rights on the data pages it caches, and a locating scheme to get to the most up-to-date version of a data page. Our problem is to recover this token directory after a site has failed. Our solution is to treat the token directory at each site as a fragment of a global token database and the page migration activities as token transactions that update this distributed database. By the use of the unilateral commit protocol (UCP) for token transactions, fast recovery of the token state at minimal run time overhead of token transactions execution can be achieved.

1 Introduction

Most distributed systems are structured as distributed processes running at different computer nodes (or sites). Each of these processes manages its own data. When a process \( p_i \) on site \( s_i \) wants to access data residing on \( s_j \), it sends a request to a process on \( s_j \) to carry out the operation and return the result, if any, to \( p_i \). This approach is used in systems such as CAMELOT [Spec 87], R* [Lind 82] and ARGUS [Link 84]. Remote procedure calls [Nels 81] are often used to implement such remote requests.

The abstraction of distributed shared virtual memory (DSVM) is an alternative and complementary paradigm which integrates the memory space of all the sites into a single logical entity shared by cooperating processes executing on multiple sites. [Bell 90] [Delp 88] [Fori 89] [Hsu 89] [Li 86] [Rama 88]. The idea is to treat the main memory of the individual sites as cache such that a process's address space can span machine boundaries. When a process on a machine needs to access data which is not currently resident in its local storage (main memory and backing store), a network page fault is generated and the page is migrated from a remote site. The underlying virtual memory system must be able to correctly locate the most up-to-date version of a data page as the page migrates in the system and as new versions are created or cached. We use the term token table to refer to a site's knowledge as to what access rights it has on data pages it has cached, and the term locating table for the information on how to get to the sites which have the most up-to-date data pages. Together, they form the token directory. Much of the recovery problem centers on the recovery of the token directory of a failed site. Without precautions, a recoverying site may be required to consult every other site's token table and deduce from them the tokens it actually owns. This was actually suggested in [Mino 82]. In situations where the token table is small and fast recovery is not necessary, this solution may be acceptable. However, in a DSVM system with a very large address space and thus a large token table, this method will be very expensive.

Our solution to the recovery problem is based on the method of token transaction proposed in [Tam et al. 90]. The idea is to regard data migration activity as a special kind of transaction that updates a persistent "token database", namely, the token directory. By updating the token directory in a "pseudo atomic" fashion, the token directory state will retain a certain level of consistency in face of failures. This consistency can be exploited to allow the recovery algorithms to be simplified and made more efficient. The main contribution of the paper consists of an abstraction and an extension of the token transaction framework, the use of token transactions to achieve fast recovery in a DSVM system, and a study of the token transaction execution overhead in a DSVM system.

The paper is organized as follows. In the remainder of this section we briefly describe different locating schemes and other related work. Section 2 introduces the basic model of our DSVM system. Section 3 describes a generic token transaction system; it provides an abstraction of the system first proposed in [Tam et al. 90], and the latter can now be seen as a particular example. In Section 4 we define a token transaction system for managing page migration in our DSVM system, and describe how fast recovery can be achieved. Section 5 is a performance study of the runtime overhead (primarily a logging overhead) of executing token transactions in a DSVM system. Section 6 summarizes the paper.

1.1 Different Page Locating Schemes for DSVM

As discussed above, a DSVM must provide a page locating facility. The locating schemes for DSVM systems can be roughly classified into broadcast-based schemes and directory-based schemes.\(^1\)

\(^1\)It is interesting to note the analogy of this classification to that of the cache consistency schemes in multiprocessor systems: snoopy cache and the
The broadcast-based locating scheme [Hau 88] does away with a locating table, but it requires every network page fault to result in a broadcast message in the network and each site would check its own token table to decide whether and how to respond. After a failure, say, site $s_1$, $s_2$ throws away its token table and, upon recovery, faults on every page access. However, if the only up-to-date version of a page resided on $s_1$ when $s_2$ failed, then when $s_1$ recovers, an elaborate procedure which requires consultation with all other sites is needed to infer this fact and to recover the page. In general, the broadcast-based scheme does not scale up well, and we do not consider this scheme in our model of DSVM.

Directory-based schemes can be further classified into static schemes and dynamic schemes. A static scheme designates a specific site as the "locator" site for a particular data page. Every time a network page fault occurs, this site is consulted and if necessary the directory updated. A dynamic scheme traces a path of sites to find out the latest valid version of a page [Fowl 86]. Without giving the details of these schemes, we observe that all directory-based schemes share the common characteristics that the locating table is constantly updated when a directory inquiry. As failures occur, in addition to the loss of a local token table, the information in the locating table may also be inconsistent, and in general may have to be reconstructed by a protocol involving all sites.

Our approach to recovery alleviates the need for such reconstruction. Although we will present our approach based on a particular static directory-based locating scheme adopted in [Li 86] (described in detail in Section 2), the approach can be generalized and applied to other locating schemes as well.

Note also that our DSVM system does not have shared disks. A study of the recovery problem of DSVM systems which uses a central shared disk was presented in [Wu et al. 89]. They used a directory-based locating scheme, where a given site serves as the locator of a page, and the locating table is kept at the locator. A checkpoint is performed for process $p$ at site $s$ every time a page $d$ that was last modified by $s$ is migrated to another site. By performing a checkpoint before a page is migrated, the system guarantees that no processes can read dirty and unsecured data. This avoids the problem of cascaded abort. However, they did not provide a special mechanism for recovery of the token directory. When a site $s$ recovers from a system failure, it simply resets its token table entries to null. If $s$ is a locator, recovery of the token directory upon recovery of $s$ will have to involve all sites in the network.

2 A DSVM System Model

2.1 The Process Model

In this paper, a computation denotes an overall task, such as the execution of a parallel program. A computation consists of multiple processes, or execution threads, which share a virtual memory space $D$ consisting of pages $(d_1, d_2, \ldots, d_n)$. Data pages can be read or written. Access to the shared data space is synchronized at the process level by means of a sync-unit. A sync-unit is a unit of atomicity very much like a critical region. A process can access the shared space only inside sync-units. A sync-unit is modeled as a sequence of read and write operations on pages in $D$. A typical process profile is displayed below:

begin-sync-unit
  access and update shared data;
end-sync-unit

access and update local data;

\footnote{A checkpoint for a process $p$ is performed by writing all the dirty pages modified by $p$ back to the central shared disk and recording the necessary state information such as program counter for restart.}

Like the commit point of a database transaction, at the end of the sync-unit, the dirty data modified within the sync-unit is released. However, end-sync-unit does not necessarily guarantee data persistence (see Section 2.4 for discussion on checkpointing.)

2.2 Distribution of Shared Pages

There are multiple sites in the system. Sites do not share memory or disks. At any point in time, for every data page $d$ in $D$, one site is designated as the owner site of $d$, denoted as owner$(d)$, and $c$ sites are copy sites of $d$, where $c = 1, \ldots, m$. The set of copy sites of $d$ is denoted as copyset$(d)$. copyset$(d)$ includes at least owner$(d)$. The owner site and copy sites have certain rights to accessing $d$ locally, which will be explained later. owner$(d)$ and copyset$(d)$ are changed dynamically due to page migration.

Each sync-unit $t$ within a process executes on a single site, denoted as site$(t)$. A sync-unit $t$ can perform a read operation on $d$ only if site$(t)$ is a copy site of $d$. It can perform a write operation on $d$ only if site$(t)$ is the exclusive copy site of $d$. When the preconditions fail, the sync-unit cannot proceed with the operation, and a network page fault is generated and page migration must be performed. The sync-unit resums when the precondi-

2.3 Network Paging

To handle a network read fault, site $s$ sends a "read" request to owner$(d)$ through a locator. (locator will be explained shortly.) owner$(d)$ waits till no local write lock exists on $d$, then grants a read token of $d$ to $s$, along with a copy of $d$, thus elevating $s$ to be a copy site. owner$(d)$ also remembers that $s$ is now in copyset$(d)$.

Similarly, for a network write fault, $s$ sends an "owner" request to owner$(d)$ through locator$(d)$. owner$(d)$ waits till no local locks exist on $d$, then transfers the ownership (referred to as the owner token) and the copyset of $d$ to $s$, along with a copy of $d$ if necessary. $s$ then becomes the new owner, and sends a "read token in-validate" request to each of the sites, except itself, in copyset$(d)$. Each copy site of $d$ waits till no local read lock exists on $d$, then gives up its read token by sending it to $s$. The write precondi-

To conveniently test the preconditions of local read/write op-

operations, every site must know the collection of page tokens it owns. Furthermore, the owner of $d$ must know the copyset of $d$.

In our model, we use a simple static locating scheme. For each page $d$ there is a unique site designated as the locator site of $d$, denoted as locator$(d)$. locator$(d)$ keeps track of the current owner of $d$, and $chain(d)$; $chain(d) = s_1, s_2, \ldots, s_k$ is a sequence of sites which have requested for owner token transfer of $d$. The head of the chain is the current owner, while the tail is the site which has most recently requested a transfer. A site $s$ requesting for owner token of $d$ will send its request to locator$(d)$, which in turn sends the request to the tail of $chain(d)$, and then appends $s$ to the tail end of $chain(d)$. For any given $d$, locator$(d)$ is static and known to all sites.
2.4 Checkpointing and the Recovery Problem

Under the DSVM paradigm, a sync-unit t only holds locks locally, and it updates d in D only if site(t) is the exclusive copy site of d. Each site is assumed to have some volatile memory and a dedicated logging device. When t reaches its end, a sync-point log record for t is appended to the log buffer. A sync-point log record for t contains all the dirty pages modified by t and the necessary state information for restart. Periodically a site forces the log buffer which secures the sync-point image for t. By combining the sync-point record logging and the standard before-image write-ahead log protocol [Gray 81], the system is able to guarantee failure atomicity of sync-units executed at each site. In addition, to guard against cascaded abort, the sync-point log record for t has to be forced before a dirty page d created by t is migrated to another site.

When t fails and subsequently recovers, the failed process's image is restored to the most recent forced sync-unit t that survived on the logging device and the process can resume from that point. (see Figure 1). It is easy to see that the process's local (i.e. non-shared) space is properly updated. However, since the shared data might have been migrated to other sites, the above mechanism is insufficient to recover the state of the shared data space.

What needs to be done is to recover the site's knowledge of the collection of page tokens it owned when the failure occurred. This is the subject discussed in the following sections.

3 A Generic Token Transaction System

3.1 Basic Structure

A generic token transaction system consists of a distributed token database and a set of token transaction classes. A token transaction tt consists of a precondition followed by a sequence of operations on the token database. tt will not be executed if its precondition is not satisfied. In general, a tt will access the token database fragments on several sites, and therefore is a distributed transaction. The site on which tt is initiated is called an initiator, denoted as site(t).

A strong consistency requirement is defined for the token database, and each token transaction correctly transforms the token database from one strongly consistent state to another. Token transactions can be executed in the same way generic distributed transactions are executed, employing mechanisms such as two phase locking and two phase commit to guarantee the failure atomicity, serializability atomicity and permanence. The fact that token transactions satisfy the atomicity requirements guarantees consistency and recoverability of the token database.

What makes token transaction systems unique is the existence of a weak consistency requirement, which provides for an opportunity to execute token transactions much more efficiently. A weak consistency requirement is defined such that if the token database is weakly consistent, then no incorrect synchronization of the accesses to the object-level shared data (e.g., the shared virtual memory) by the cooperating processes may result. In other words, if the token database is weakly consistent, then the shared data cannot be corrupted. However, weakly consistent token states may imply less availability of the shared data to the cooperating processes, which is not desirable. Intuitively, weakly consistent token state represent states which can be tolerated in the short run, but a guarantee must exist to ensure that strong consistency will be regained in the long run.

Note that, depending on the choice of page migration algorithms (e.g., access modes, locating methods, availability requirements) different token transaction systems (with specific token database structures and distribution, specific classes of token transactions, and strong and weak consistency requirements) are instantiated. In Section 4, we will show how our DSVM system is mapped to a token transaction system, and the notions of strong and weak consistencies will be exemplified.

3.2 UCP Decomposition

The existence of an explicitly stated weak consistency requirement makes it possible to consider an efficient method for executing token transactions. Each token transaction can be decomposed into subunits such that each subunit maps the token database from one weakly consistent state to another and each subunit may be executed and "committed" asynchronously.

More importantly, if a token transaction can be decomposed into a number of "1-site" subunits (a 1-site subunit is a subunit which updates only token database fragments residing on a single site) then the overhead of distributed commit of the token transaction may be significantly reduced. One particular class of decomposition was considered in [Tam et al. 90] and can be generally described as follows. Token transactions are broken into a number of 1-site subunits tsub1, tsub2, ..., tsubm, such that the permissible execution sequences are those with tsub followed by any sequence of tsub1, tsub2, ..., tsubm, where tsub is a subunit which updates only the token database fragments residing on the initiation site of the token transaction, and tsub1, ..., tsubm are subunits which update those on other sites s1, ..., sn.

With such a decomposition, each subunit tsub1 can execute and "commit" independently. However, if failures occur, some subunits which should execute but have not yet been executed may be lost. The following requirement must be implemented to ensure that strong consistency will eventually be regained: tsub1, ..., tsubm are executed and committed if and only if tsub commits.

This class of decomposition is called UCP decomposition. It's been shown in [Tam et al. 90] that, in a simple data migration system which supports owner tokens only and uses the static directory-based locating method, every token transaction can be UCP-decomposed. UCP decomposition enables the use of the Unilateral Commit Protocol to execute a token transaction (thus the term UCP decomposition).

3.3 The Unilateral Commit Algorithm

Let cohorts(tt) denote the set of cohort sites (not including site(tt)) of tt. Based on UCP decomposition, tt can be decomposed into two types of subunits: (1) a 1-site subunit at site(tt), tsub1, and (2) multiple 1-site subunits at the cohort sites, each consist of remote updates on the fragment stored at cohorts(tt).

A token transaction tt commits at site(tt) after the event for it has been secured at site(tt)'s logging device but before site(tt) sends out the remote subunits to cohorts(tt). That is, tt commits unilaterally at site(tt) without waiting for the explicit agreement from its cohorts. When a remote subunit arrives at a cohort site, the cohort logs the event (but without the need to immediately force the log), and applies the updates.

If failure occurs after site(tt) "commits" a token transaction but before the other cohorts receive the messages regarding this token transaction, a recovery protocol makes sure that the missing messages will be executed at the cohorts upon recovery. This is achieved through an underlying reliable capture mechanism which resembles a reliable communication protocol. Since all remote subunits are logged at the initiator, retransmission can...
be used should the cohort site misses the message due to failure. To detect lost and duplicated messages, the algorithm uses a message sequence number. The complete UCP algorithm can be found in [Tam et al. 90]. The unilateral commit protocol requires fewer messages than standard two-phase commit (2PC) when running a distributed transaction, and there is less message latency during commit (For a comparison between UCP and 2PC, see Figure 2).

4 Managing DSVM using a Token Transaction System

As described in the previous section, a generic token transaction system consists of a token database, token transaction classes, and specification of strong and weak consistency requirements. We map our DSVM system defined in Section 2 to a token transaction system as follows: we will define the token directory as the token database, specify a set of token transaction classes which handle migration and locating of read and owner tokens, and define strong and weak consistency requirements. The design is an extension of that illustrated in [Tam et al. 90] (the extension is primarily to handle read tokens in our DSVM system, and necessitates the addition of three different token transaction classes), and this extended token transaction system still allows each token transaction class to be UCP decomposed. Thus the token directory in our DSVM system can be managed by token transactions executed using the Unilateral Commit Protocol.

4.1 Token Directory

We model the token state for D as the state of a distributed persistent token database, and the state transitions as atomic updates to this database. The token database consists of three data structures: tokenable, chaintable, and forwardtable.

tokenable represents the token table and has four fields: site, page, token and copyset. tokenable is partitioned according to the site field.

chaintable has two fields: page and chain. chaintable is partitioned according to the page field and is kept at locator(d).

forwardtable represents the forward requests table. Each entry in forwardtable has three fields: site, page and forward. forwardtable represents the next site to which s should forward its owner token for d once it is done with it. forwardtable is partitioned according to the site attribute.

Finally, each site s keeps a table of flags invalidating to indicate if it is in the process of invalidating the read copy sites for a particular page. The flag for d is reset when all of d's outstanding read copy requests are collected by s. Note that the invalidating table needs to be persistent. The tradeoff is discussed in Section 4.3.

At system initialization, for all d in D, locator(d) is the owner and the exclusive copy site for d, the forward pointer for d is null; and locator(d) is the only site in chain(d).

The token database is updated only by token transactions.

4.2 Token Transactions for Network Paging

There are five classes of token transactions:

1. chain-transaction(d, data)

2. owner-token-transfer(d, si_sj)

3. distribute-read-token(d, si_sj)

4. return-read-token(d, si_sj)

5. forward-read-token(d, si_sj)

The chain transaction, initiated at locator(d) in response to an owner request from site sj, appends the transfer request to the tail of chain(d), say s;

Precondition: sj is not already in chain(d)
begin.transaction
append sj to tail of chain(d) in T /* local update */
add s to sj's forward field for d in T /* remote update */ end.transaction

The precondition guarantees chain(d) will not be circular.

2. owner-token-transfer(d, si_sj)

The owner-token-transfer transaction, initiated at site si, transmits the owner token for d from site si to site sj;

Precondition: the forward link (d, si, sj) is in T, si has the O token and si's invalidating flag for d is null.
begin.transaction
delete si from sj's forward field for d in T /* local update */
delete sj's O token for d in T /* local update */
remove-read-token(d, si, sj, C) /* remote update */
insert O token and copyset d into sj's token table /* remote update */
end.transaction

The precondition guarantees that site sj can only transfer the owner token if it is not in the middle of collecting the outstanding read copy tokens. This reduces the copyset chaining discussed in Section 4.3.

3. distribute-read-token(d, si_sj)

The distribute-read-token transaction transmits a read token (R) from the owner site si to site sj in response to a read request.

Precondition: si has the O token
begin.transaction
add s to copyset(d) si; sj /* local update */
insert R token for d into sj's token table /* remote update */
end.transaction

4. return-read-token(d, si_sj)

Initiated at the copy site si, a return-read-token transaction returns the read token to the owner sj in response to a copy invalidate request.

Precondition: sj has the R token
begin.transaction
delete sj's R token for d in T /* local update */
remove-copy(sj, d, si_T) /* remote update */
end.transaction

5. forward-read-token(d, si_sj)

Initiated at si, a previous owner site for d, a forward-read-token transaction forwards the read token to the new owner sj after sj receives it. The definition of forward-read-token transaction is the same as the return-read-token transaction.

/* remove-read-token(d, si, sj) */
/* remove-copy(sj, d, si_T) */
The precondition of a token transaction $tt$ can be checked locally at site$(tt)$. Each token transaction is a distributed transaction.

### Contending Token Requests

The forward-read-token transaction is devised in light of the following situation. After $d$'s owner site $si$ has sent out the "read token invalidate" requests to the other sites in copyset$(d)$, $si$ could transfer the owner token for $d$ to $sj$ in response to $sj$'s owner request. By the time the read tokens are received by $si$, $si$ is no longer the owner. To correctly update the copyset field, $si$ can start a forward-read-token transaction to forward the $R$ token to $sj$. This forwarding action may continue if $sj$ has since transferred the $O$ token to $sk$. But eventually this dangling $R$ token will catch up the latest owner who would update the copyset field correctly.

To minimize this chasing, a site should not give up its $O$ token once it has started the invalidating process. The flag invalidating serves this purpose. Had we made the invalidating table persistent, the forward-read-token transaction would not be needed. The provision of this token transaction class gives more flexibility for a site to handle contending token requests.

### Strong and Weak Consistency

**Definition 1:** Strongly consistent state: A token state is strongly consistent if $\forall d \in D$

1. there is exactly one owner$(d)$ in $T$,
2. chain$(d)$ presents a linear chain,
3. owner$(d)$ is the head of chain$(d)$, and
4. if a forward pointer exists between $si$ and $sj$, then $sj$ must follow $si$, in chain$(d)$, and
5. $si$ has a $R$ token for $d$ if and only if $si$ is in copyset$(d)$.

Conditions 2 and 3 guarantee that the locating information is accurate. Condition 4 guarantees that the forwarding action is not broken. Executing a token transaction using 2PC transforms a strongly consistent token state to another.

One observes that if the transient token state between the 1-site subunits of our token transactions is revealed to the outside, the underlying shared memory space $D$ is not corrupted. For instance, after the local updates of an owner-token-transfer transaction is applied but before the remote updates appear on the receiver site, the fact that an owner token in transit is missing from the token state will not affect the correct synchronization of access to data pages in the DSVM.

**Definition 2:** Weakly consistent state: A token state is weakly consistent if $\forall d \in D$

1. there is at most one owner,
2. chain$(d)$ presents a linear chain,
3. the owner is one of the sites in chain$(d)$, and
4. if a forward pointer exists between $si$ and $sj$, then $sj$ must follow $si$, in chain$(d)$, and
5. if a site $s$ has a $R$ token, $s$ must be in the copy set attribute at the owner site.

Although weakly consistent states are acceptable, the eventual progress of the underlying database still depends on the maintenance of a strongly consistent state. In the example above, the data page will be unavailable until the token that is in transit arrives at the destination site. It can be shown that the above specific design of token transaction system for our DSVM system can be UCP decomposed. Therefore Unilateral Commit Protocol is used to execute token transaction.

### 4.5 Fast Recovery of Token Directory

A site periodically checkpoints its token state along with the sequencing numbers to stable storage. When a failed site $si$ recovers, $si$ first restores the last token directory checkpoint and rolls forward the local token state based on the remaining log entries saved in the local log. This completes the site's local recovery. New processes or threads can now be started on $si$, and the local token table fragment of $si$ allows these processes to immediately access data pages cached at $s$, before $si$ failed. This gives these processes on $si$ a "warm" start. $si$ now also joins the network and participates in normal page migration activities.

If there were lost $tt$ update messages between $si$ and other sites after $si$ has failed, these messages will be recovered during the normal course of interactions between $si$ and other sites. However, to speed up the detection of these lost update messages, (i.e., to allow for a faster convergence to a strongly consistent token state), $si$ can take the initiative to broadcast its message sequence numbers to other sites that are up, and solicit from all other sites their sequence numbers. After the broadcast, the normal procedure for $tt$ update message processing will automatically ensure that all missing messages are recovered.

As readily observed, the local recovery is as fast as restoring the token directory checkpoint and rolling forward the log. In comparison with previous schemes in which the token directories of all the other sites have to be assembled in order to reconstruct the failed portion of the directory information, our recovery protocol is much faster. Furthermore, our method is incremental and takes advantage of the checkpoints image of the token state. The recovery time is limited by the volume of network paging activity since the last checkpoint interval. Our method is particularly attractive when the system tends to encounter frequent but short-duration failures and when fast, "warm" restarts are desirable. In other schemes, a complete reconstruction of the token directory is often needed, resulting in slow and "cold" restarts.

### 5 Overhead of Managing DSVM with Token Transactions

Using a simulation model similar to the one in [Bell 90], we study the performance impact of logging due to token transactions.

#### 5.1 Workload Characterization

Our model runs sync-units on a network of machines. We characterize the workload of the sync-units simply as a series of computations on the shared address space. Each unit accesses and processes $K$ different data pages. Access to the data pages are subject to two-phase locking. Once the page request has been granted it will be processed for an average of $T_p$ time units by the sync-unit, before the next request for a data page is generated. $T_p$ is called the data granule processing time. Every time a token transaction is executed, UCP logging is performed. To avoid cascaded abort, we spoil the dirty pages of a sync-unit to the log at end-sync-unit, and the log containing a dirty page $p$ is forced before $p$ migrates.

The database consists of $N$ pages. Each site has a fixed number of sync-units running concurrently. This number is the multi-programming level of the site (MPL).

#### 5.2 System Architecture

We model a distributed environment consisting of a fixed number of machines linked over a local area network. Each site is assumed to have two cpu's, an abundant amount of main memory that can hold the entire shared address space, network communication channel processors and a dedicated logging device. We model contention for the different hardware resources as follows.

Logging requests are simply queued and handled in a first in, first out manner. The total logging time is divided into three
stages: (1) the time to write the log to a log buffer; (2) the time to clear the log buffer and write the data onto the disk; and (3) interrupt handling. Stage 2 is actually composed of disk access time, set by the \( T_{disk} \) parameter, and disk transfer time. We study two models of logging: serial logging in which the disk is started every time some data is to be logged; the "train model" in which the data is accumulated in the log buffer and the disk is only started periodically. This period is determined by \( T_{disk} \). In both cases, the process that is running the token transaction gets blocked until the logged data is written to the disk.

CPU's are scheduled in a round-robin fashion. In addition to processing data granules, message processing also contends for processors.

We use two parameters to define the raw message delivery time (excluding queuing delay). We assume that there is a fixed delay \( D_f \) for transmitting a message, and a variable delay proportional to the size of the message. This variable delay is calculated as \( R_v \times \frac{s}{T_v} \) where \( s \) is the message size in message units. For this simulation we assume that a message unit is the same size as one data page. Thus, average time required to send a message \( s \) units long is equal to \( D_f + R_v \times \frac{s}{T_v} \).

The metric we use in this performance analysis are:

- \( t \): throughput in sync-units per unit time \( T_p \)
- \( r \): sync-unit response time in unit time \( T_p \)

\( t \) is the total number of finished sync-units divided by the elapsed simulated time (in units of \( T_p \)) of the system. Since token transactions only incur the overhead of logging, the other performance metric such as network bandwidth requirement is not affected. These measurements can be found in a previous study [Hsu 89].

### 5.3 System Parameters

The following system parameters were in effect during all the simulations, unless otherwise noted. The parameters that vary the most are the logging device specifications.

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>( N )</td>
<td>10,000</td>
</tr>
<tr>
<td>( MFL )</td>
<td>2</td>
</tr>
<tr>
<td>( K )</td>
<td>8</td>
</tr>
<tr>
<td>( W )</td>
<td>25%</td>
</tr>
<tr>
<td>( B_f )</td>
<td>( T_v + R_v )</td>
</tr>
<tr>
<td>( R_v )</td>
<td>( T_v )</td>
</tr>
</tbody>
</table>

Every page write updates an average 120 bytes. To log a token transaction takes 40 bytes. We implicitly consider \( T_p \), the cpu processing time for each page, to roughly correspond to 3ms. The memory to log buffer transfer time is 0.0007T, per byte, which corresponds to 300ns per byte. The disk throughput rate is 0.00035T, per byte, which amounts to 1MB per second. Disk interrupt service time is 0.333T, or 1ms.

\( R_f \) and \( R_v \) are set so that it would take an average of 6ms = \( R_f + \frac{1}{T_v} \times R_v \) to deliver a one-page message, or 12ms for a round-trip delay on a lightly loaded system. This is roughly consistent with recent studies on the performance of the UDP protocol in the local area networks (e.g., [Cabr 88]).

### 5.4 Simulation Results

Figure 3 summarizes the system throughput by varying the logging device capability. Throughput rises with number of sites. It shows the ideal throughput of 2.12 for site = 100 when there is no logging involved. However, in order to perform any kind of recovery, periodic checkpointing of the shared data is necessary. Therefore the reference line to compare the overhead of UCP logging is the line in bold that attains a throughput of 2.05.

With the serial logging model for token transactions that incur a disk access time of \( 3T_v \), the degradation is noticeable (1.15). This is due to the significant delay of a disk force every time a token transaction is run. With the train model for logging, however, throughput is 1.82 or about 11% lower than the reference line when the logging device is modestly efficient (\( T_{disk} = 0.337T_v \) and \( T_{disk} = 1T_v \)).

Figure 4 shows the average number of token transactions executed per sync-unit. Figure 5 shows similar behavior for the response time metric.

### 6 Summary and Future Work

In this paper we have argued that token transactions can be used to reliably and efficiently manage the transitions of the token state in a DSM system by the use of the unilateral commit protocol (UCP). Given a persistent and atomic token directory, recovery of the token state after a system failure is as fast as rolling forward a system checkpoint. Based on simulation results, we have shown that the run time overhead of the UCP is relatively small. One direction for future research is to reduce the amount of synchronous log force at the initiator of a token transaction. Our present performance study is based on the site's having a dedicated logging device. Another direction is to study the impact if this requirement is relaxed.

We have demonstrated our ideas using a static directory-based locating method. The generic token transaction system can be applied to DSM systems using other locating methods, such as the dynamic scheme used in [Fowl 86]. The dynamic scheme may require a development of a class of token transaction decomposition which is more generalized than the current UCP decomposition, and may provide insight into how the framework for "token transaction decomposition" can be applied to general transaction systems.

### References


write sync point log record forced before \( d \) transmitted to \( s_3 \)

After recovery, process restarts here.

Figure 1: Computation Model

- \( s_1 \)
- \( s_2 \)
- log buffer for \( s_4 \)
- sync point record for \( s_4 \)
- sync point record for \( s_3 \)
- sync point record for \( s_2 \)

Assume log buffer to the left forced before failure, at recovery sync point of \( s_2 \) is restored for \( s_3 \).

Figure 2: Executing token transactions using 2 Phase Commit vs. Unilateral Commit