プログラミング−言語・基礎・実践−　5−13
(1991. 11. 22)

マルチプロトコルによる分散タプルスペースの実現

千葉 滋　　加藤 和彦　　益田 隆司
東京大学 理学部 情報科学科

あらまし　　マルチプロトコル・タプルスペース (MTS) とはタプルスペースを分散環境に実装したものである。タプルスペースはさまざまな種類の通信やデータ共有の記述を容易にするプリミティブをもつ。しかしながら共有メモリ型の計算機上での実装と異なり、分散環境上では単純な方法では実行効率のよいタプルスペースを実現することはできない。MTS は複製技術を利用してタプルスペースを実現し、複数の一貫性保持のためのプロトコルを提供する。これらのプロトコルは弱い一貫性保持の方法を使って、複製の一貫性保持の効率化を通信のパターンに応じておこなう。

Exploiting Multiprotocol to Implement Distributed Tuple Space

Shigeru CHIBA　　Kazuhiko KATO　　Takashi MASUDA

Department of Information Science, University of Tokyo
7-3-1 Hongo Bunkyo-ku TOKYO 113

Abstract　　Multiprotocol Tuple Space (MTS) is a distributed implementation of Tuple Space. Although Tuple Space provides primitives convenient for describing several types of communication and data-sharing, its naive implementation in distributed environments is less efficient than the implementation on shared-memory machines. MTS is a replication-based implementation and provides several different replication-maintenance protocols that exploit weak consistency. These protocols achieve efficient replication-maintenance according to communication patterns.

1 Introduction

As distributed computing systems advance, numerous distributed applications are being developed. They run on workstations connected to networks such as Ethernet. For the natural modeling of the distributed applications, the programmer needs application specific communication form and data-sharing scheme. For example, these applications require various forms of communication and data-sharing schemes such as remote procedure calls, streams, and sharing many kinds of data (integer arrays, graph structures, etc.) The programmer should choose the most appropriate scheme from them. Because it is difficult to combine many communication facilities in one application, it is preferred that the system provides a single flexible facility, which can model many communication forms and data-sharing schemes.

One approach to provide the flexible facility is to implement distributed shared memory (DSM) [14]. It is used in the same manner as ordinary shared memory in a tightly coupled multiprocessor machine. It is, however, not easy to describe communication through DSM because it requires complex synchronization. Furthermore, DSM uses strict consistency, in which the value returned by a read operation is up-to-date. This hinders efficient implementation of DSM because up-to-date values do not naturally exist in distributed environments. Several works have introduced relaxed consistency [2, 8] or much weaker consistency [7] into DSM. In relaxed consistency, a memory is consistent if there are some legal schedules of operations. Relaxed consistency is identical to strict consistency from the viewpoint of a programmer. Weak consistency is substantially different from strict consistency. Weak consistency provides new semantics for DSM. These systems, however, have problems. The former needs information about application semantics. The latter makes programmers conscious of the different consistency.

Another approach is to use Tuple Space [6], which allows programmers to describe several types of communication and data-sharing [4, 13, 15]. It can use relaxed consistency because atomic operations of Tuple Space are synchronous while read and write operations of DSM are asynchronous. Because relaxed consistency allows different schedules for operations, we can schedule the operations to execute them most efficiently. Multiprotocol Tuple Space (MTS) [5], which this paper presents, is a distributed implementation that exploits this benefit. It is a replication-based implementation and provides several protocols for maintaining replications. These protocols are weakened to different degrees. They are selected while keeping the consistency, and improve the execution speed of Linda programs [6] (Linda is a language that has Tuple Space operations as its primitives.) This is because Tuple Space can use relaxed consistency. Naive implementations cannot sufficiently exploit the benefit of Tuple Space. If a central server is responsible for access to Tuple Space, it becomes a bottleneck. If we simply use replications that are maintained by a single protocol, keeping them consistent decreases efficiency. MTS improves efficiency by using multiple protocols.

This paper presents MTS and discusses how consistency is maintained by each protocol that MTS provides. Then, it shows typical examples in which the protocols are selected according to communication patterns.

2 Multiprotocol Tuple Space

MTS is a replication-based implementation of Tuple Space in distributed systems, such as workstations connected via Ethernet. It does not require that the underlying network has hardware support for multicasting. Each node (workstation) has a replication of Tuple Space. There is no central server responsible for updating the replications, but any node can modify a state of Tuple Space and update the replications, using reliable point-to-point messages. It is assumed that messages are delivered in the order in which they are sent. The replications are updated according to the protocols. MTS provides several protocols that guarantee weakened replication-consistency, as well as a protocol that guarantees strict consistency. This section briefly describes Tuple Space and the protocols of MTS.

2.1 Tuple Space

Tuple Space is global memory shared by processes running in a parallel environment. It was proposed for the distributed language Linda [6]. Tuple Space contains *tuples*, which are lists of typed fields. For example, a tuple `("Tokyo", 3)` consists of a string field and an integer field.

A process manipulates tuples in Tuple Space by three atomic operations, here called *TS operations*. `Out` generates a new tuple in Tuple Space, `Read` reads a tuple, and `In` removes a tuple from Tuple Space. Because `In` reads a tuple and simultaneously deletes it, a tuple is never read after it is removed by `In`. Tuple Space is addressed associatively. `In` and `Read` remove or read a tuple that matches their templates. For example, the following operation

```
In("Kyoto", ?i, ?j)
```

removes a tuple whose first field is the string "Kyoto" and whose second and third fields have the same types as the variables i and j. If there is a tuple matching this template, that tuple is removed. Its second field is assigned to i, and its third to j. If there are several matching tuples, In chooses one of them arbitrarily. In and Read wait for a newly-generated tuple if no tuple matches their templates.

The original Tuple Space includes other operations: Eval, Readp, and Inp. Eval generates a new process. Because process generation is an issue indifferent to interprocess communication, this paper does not deal with Eval. Readp and Inp are nonblocking operations: if no tuple matches their templates, these operations fail and do not block. This paper does not deal with them because they are extended operations and not essential parts of Tuple Space. Such nonblocking operations, which inspect the "present" state of Tuple Space, are inappropriate for distributed environments, where the most recent operation is not defined.

2.2 Description of Protocols

Of the three TS operations, only Read requires no update of replications. A node in which a Read is executed searches its local replication and does not modify the replication. On the other hand, Out and In require updates of replications. A node in which they are executed must pass the result of these operations to all replications. The node sends messages to other nodes and all replications are updated according to protocols. MTS provides one protocol for Out and three protocols for In.

When a tuple is generated at a node P, the node P multicasts 'out' messages to the other nodes. The nodes that receive those messages update their own replications. No reply messages are returned to node P. One execution of Out requires one multicasting.

Next, we look at the three protocols: the *strict* protocol, the *nonexclusive protocol*, and the *weak* protocol. These are shown in Figure 1. They guarantee replication-consistency with different degrees of weakness. The next section discusses the consistency of Tuple Space in detail.

The strict protocol guarantees that replications are strictly consistent. It is based on the 2-phase lock algorithm and makes In complete after a tuple is removed from all replications. Messages are multicast twice and reply messages are returned after the first is received.

When a node P attempts to remove a tuple, it multicasts 'lock' messages to the other nodes in order to lock the tuple. The nodes receiving these messages reply with an 'accept' message if the tuple has not yet been locked or has been locked by a node without priority than node P; otherwise those nodes reply with a 'refuse' message. If all replies are 'accept', then node P removes the locked tuple from its local replication, multicasts 'remove' messages, and executes the next step. The nodes receiving 'remove' messages remove the tuple from their own replication. If one of the replies is a 'refuse' message, then the removal fails and node P tries to remove another tuple (*rollback*).

An In that is executed according to the nonexclusive protocol is not mutually exclusive. The nonexclusive protocol keeps consistency only when two In's never try to remove the same tuple simultaneously. Other Read's can try to read it simultaneously. Although this protocol guarantees weaker consistency, it requires fewer messages than the strict protocol. In the nonexclusive protocol, messages are multicast once and replies to them are returned.

When a node P attempts to remove a tuple, it multicasts 'delete' messages to the other nodes and removes the tuple from its local replication. The nodes receiving those messages remove the tuple from their own replications and return replies. Note that this protocol assumes that no conflict of In's occurs. Node P executes the next step after receiving all replies.

The last protocol is the weak protocol, which guarantees the weakest replication-consistency and requires the fewest messages. In this protocol, messages are multicast once and no reply is returned. A node that attempts to remove a tuple multicasts 'erase' messages to the other nodes and then removes the tuple from its local replication. The nodes receiving those messages also remove the tuple from their own but do not reply to the 'erase' messages.

The weak protocol only informs the other nodes of the removal of a tuple. It is, hence, not mutually exclusive. This protocol executes an In and goes onto the next step before a tuple is removed from all replications, so the weak protocol cannot guarantee the consistency when other Read's try to read simultaneously the same tuple that the In removes. This will be discussed again later.

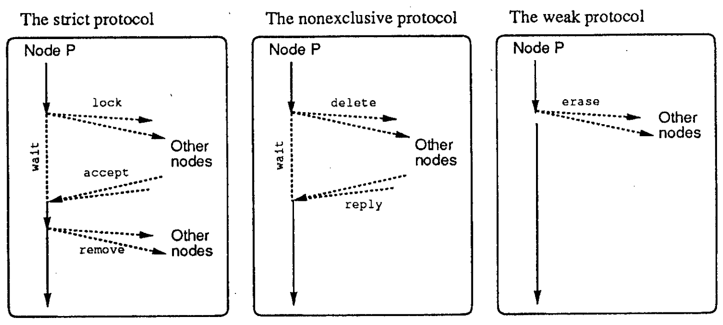

Figure 1: Three Protocols for In

3 Consistency of Tuple Space

Tuple Space is appropriate for communication facilities in distributed systems. Its semantics suits a distributed environment, where it is not known which event is the most recent. This section first discusses consistency of distributed shared memory systems and compares it with the consistency of Tuple Space. Then we formally show the consistency of Tuple Space and the characteristics of the protocols of MTS.

Typical distributed shared memory systems [14] use strict consistency, in which the value returned by any read operation is up-to-date. This consistency allows only one node at a time to perform a write operation to an object, and a replication (a memory page) is updated or invalidated as soon as a write operation is done. It requires lots of message exchanges between nodes to decide which node can perform a write operation. When a node repeatedly reads data which another node writes frequently, data may be transferred from the write-node to the read-node every read operation. Enforcing strict consistency, hence, decreases the efficiency of the system.

Several methods have been proposed to avoid the inefficiency of strict consistency, Munin [2] provides a *delayed update queue*, which uses relaxed consistency. In relaxed consistency, a memory is consistent if there are some legal schedules that satisfy a causal relationship between read and write operations. Because a causal relationship is a partial order in distributed systems, this consistency allows the system to delay updates of replications. With the delayed update queue of Munin, updates are queued and do not propagate until synchronization occurs. This makes it possible to combine several updates into the same network packet. The delayed update queue, however, can lose consistency if synchronization does not occur correctly. When operations executed on the same node between one synchronization and the next are both write and read operations, the consistency may be broken. Figure 2 illustrates an incorrect synchronization. $W(X)$ means a write operation, and $R(X)$ means a read operation on an object X. P and Q are nodes; P executes $W(A), W^{\star}(A), R(B)$, and $W(C)$ sequentially, and Q executes $W(B), W^{\star}(B), R(A)$, and $W(D)$. If the first synchronization is after $W(A)$ and $W(B)$, and the second one is after $W(C)$ and $W(D)$, the consistency will be broken. $R(B)$ reads the value written at $W(B)$ and $R(A)$ reads the value written at $W(A)$ because $W^{\star}(A)$ and $W^{\star}(B)$ are still queued. $R(B)$, therefore, precedes $W^{\star}(B)$ and $R(A)$ precedes $W^{\star}(A)$. There is no legal schedule that satisfies this causal relationship.

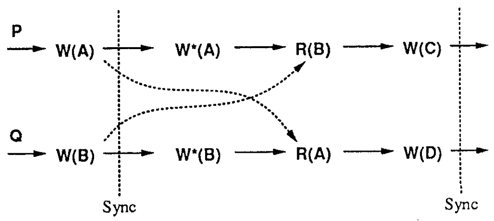

Figure 2: Delayed Update Queue

Introducing relaxed consistency into a distributed shared memory system, whose atomic operations are read and write, raises the difficulty of deciding when replications should synchronize. TORiS [8] solves this problem by exploiting an optimistic concurrency control of transactions in database systems [9]. It treats several read and write operations as a transaction, and updates by write operations do not propagate until the transaction is committed. A synchronization occurs at a commit operation. When a transaction is committed, the validity of updates (whether they break consistency) is checked. If the updates are invalid, the transaction is aborted and the program execution is rolled back. A defect of TORiS is that an application must be responsible for restoring the local state of a program when a rollback occurs.

These implementations that exploit relaxed consistency have the defects discussed above. This is because the read operation in distributed shared memory systems is asynchronous. It gets the most recent value of an object, but the most recent value is not defined exactly. The result of a read operation depends on the global state of the distributed shared memory rather than the local state.

On the other hand, it is straightforward to implement Tuple Space exploiting relaxed consistency. The `Read` operation in Tuple Space is synchronous, and a tuple read by `Read` is chosen arbitrarily when several tuples match the template of that `Read`. Hence, `Read` can be executed with incomplete local information, that is, even if its local replication does not contain all tuples and it chooses a tuple from the subset of all tuples, it is correctly executed with respect to the semantics of Tuple Space. Each node can execute a `Read` even when its replication does not contain all tuples. The result of `Read` does not depend on the global state. The semantics of Tuple Space matches relaxed consistency. The implementation this paper presents, MTS, exploits this advantage to carry out Linda programs efficiently. MTS maintains relaxed consistency with the least number of message exchanges. Some protocols that MTS provides guarantee more relaxed, weak consistency. MTS uses the protocols together in executing a program and maintains relaxed consistency.

3.1 Definition of the Consistency of Tuple Space

To represent causal relationships between TS operations, we defines a ***history***. A history of TS operations is an irreflexive partial order with an ordering relation $\prec$ defined as follows:

1. Let A and B be TS operations. If A and B are executed sequentially and A precedes B, then $A \prec B$.
2. Let A_{out} be `Out`, and let B be `Read` or `In`. If B manipulates a tuple generated by A_{out}, then $A_{out} \prec B$.

The first condition specifies a causal relationship between two TS operations that are successively executed in the same node, while the second one specifies internode communication through Tuple Space. The definition of the relation $\prec$ is similar to the "happened before" relation [11].

The specification of Tuple Space intuitively suggests that Tuple Space is consistent if:

1. An `Out` precedes a `Read` and an `In` that manipulate a tuple generated by that `Out`,
2. An `In` is never followed by a `Read` that reads a tuple removed by it, and
3. Two different `In`'s do not remove a tuple generated by the same `Out`.

To discuss the consistency of a history, we define a *sequential* history. This is a history in which the ordering relation $\prec$ is a total ordering. From the intuitive conditions, it is derived easily that a sequential history is consistent if it satisfies the following condition:

Condition 1: Let A_{in} be `In` and let B be `Read` or `In`. If B follows A_{in} (i.e. $A_{in} \prec B$), then B does not manipulate the same tuple as A_{in}.

Note that a history always satisfies the first of the intuitive conditions. Because a history is an irreflexive partial order, if $A_{out} \prec B$ (B manipulates a tuple generated by A_{out}), then $B \not\prec A_{out}$ (A_{out} does not follow B). A consistent sequential history is a legal schedule of TS operations. We define a history to be consistent if there is a legal schedule of TS operations. This definition of the consistency is a relaxed consistency. This is formally defined as follows:

Consistency: A history H is *consistent* if there is a consistent sequential history S such that $\prec_H \subseteq \prec_S$.

It can be determined whether a given history is consistent by using a *Tuple Space operation graph.* This represents causal relationships that a history contains and the relationships that a consistent history can hold. A Tuple Space operation graph $TG(H)$ for a given history H is a directed graph whose nodes are TS operations in H and whose edges satisfy the following conditions: For all edges $A \rightarrow B$,

1. the history H contains $A \prec B$, or
2. A and B manipulate the same tuple and B is `In`.

The second condition means that an `In` should follow any `Read` that read the same tuple removed by the `In`. If A and B manipulate the same tuple and both A and B are `In`'s, then $TG(H)$ includes edges $A \rightarrow B$ and $B \rightarrow A$ from the second condition. Hence, if $TG(H)$ is acyclic, there is a consistent sequential history corresponding to a given history H. The following proposition shows a way to determine whether a given history is consistent, and also shows a condition for a history to be consistent.

Proposition. A given history H is consistent if and only if $TG(H)$ is acyclic.

Proof: If $TG(H)$ is acyclic, then this graph can be topologically sorted by the ordering relation $\prec$. The sorted ordering S satisfies the expression $\prec_H \subseteq \prec_S$ because $TG(H)$ contains all edges of a history H. The ordering S is a consistent sequential history because Condition 1 is satisfied from the definition of $TG(H)$ and the fact that $TG(H)$ is acyclic. The history H is, therefore, consistent.
Conversely, suppose that a given history H is consistent, that is, there is a consistent sequential history S corresponding to H. If there is two TS operations A, B that manipulate the same tuple and B is `In`, then $A \prec B$ in the sequential history S. Hence, $B \not\prec A$ in the history H. The graph $TG(H)$ is, therefore, acyclic.

3.2 Replication Consistency Protocols

To keep the consistency discussed above during program execution, updates of a replication by `Out` and `In` have to be propagated to all nodes according to protocols. MTS provides several protocols; each protocol restricts the behavior of a program execution to differing degrees. For example, the strict protocol for `In` prevents a tuple from being removed by two different `In`'s.

To represent a restriction by a combination of the protocols in MTS, a *feasibility graph* $G(H)$ is defined for a history H. I, J, and K are sets of `In`'s that use the strict, nonexclusive, and weak protocols, respectively. Nodes of $G(H)$ are TS operations in H, and its edges are $P \rightarrow Q$ such that:

1. $P \prec Q$ in the history H,
2. Q uses the strict protocol ($Q \in I$) and P, Q manipulate the same tuple; or
3. P is `Read`, Q uses the nonexclusive protocol ($Q \in J$), and P, Q manipulate the same tuple.

If an underlying network system preserves the order in which messages are sent (MTS protocols assume this), a number of edges $P \rightarrow Q$ are added to the feasibility graph $G(H)$. Q is an `In` that uses the weak protocol ($Q \in K$). Let R, S be TS operations executed in the same node as P, Q, respectively, and suppose that R follows P (i.e. $P \rightarrow R$) and S precedes Q (i.e. $S \rightarrow Q$).

4. If the feasibility graph includes $R \rightarrow S$, the edge $P \rightarrow Q$ is added to the graph.

If a history H is feasible, then the feasibility graph $G(H)$ is acyclic. The protocols prevent a program from executing in a way corresponding to a cyclic feasibility graph. Because the protocol for `Out` puts no restriction on the behavior, there is no item describing this protocol in the definition of a feasibility graph. The strict protocol guarantees that an `In` is executed exclusively and that the `In` is completed after all replications are updated. Thus, an `In` P follows any other TS operation Q that manipulates the same tuple. Because a feasible history does not include an edge $Q \rightarrow P$, the feasibility graph is acyclic. The nonexclusive protocol guarantees only that an `In` is completed after all replications are updated. An `In` P follows any other `Read` Q that reads the same tuple. A feasible history, hence, does not include an edge $Q \rightarrow P$ if Q is a `Read`. It may include an edge $Q \rightarrow P$ if Q is `In`. Thus, if a history is feasible, the feasibility graph is acyclic. Although the weak protocol gives no restriction, it indirectly restricts the program execution because the message derivation order is preserved. Suppose that an `In` P was executed at a node according to the weak protocol. If that node executes another TS operation and sends a message to another node, then the message sent by P must arrive there before the latter message arrives. After this arrival, any `In` or `Read` Q do not manipulate the tuple that P removed. Histories are, hence, infeasible if Q manipulates this tuple after that arrival. The feasibility graphs that correspond to feasible histories, therefore, do not include an edge $Q \rightarrow P$.

The consistency of Tuple Space is kept during a program execution if a given combination (I, J, K) satisfies the following condition:

Condition 2: $TG(H)$ is acyclic for all histories H whose feasibility graph $G(H)$ is acyclic.

Because a history is feasible if its feasibility graph is acyclic, Condition 2 means that, for any feasible history H, $TG(H)$ is acyclic, that is, the behavior of the execution is consistent. To keep the consistency, the combination of the protocols has to satisfy Condition 2. MTS improves communication through Tuple Space by using as weak protocols as possible under Condition 2.

Condition 2 proves that the strict protocol guarantees the consistency in any case. If any In uses the strict protocol, the program is never executed inconsistently. Because $TG(H)$ becomes equivalent to $G(H)$ when all Ins use the strict protocol ($J, K = \emptyset$), feasible histories are also consistent.

4 Discussion about Protocols

Selecting the protocols allows us to optimize the execution of Linda programs in distributed systems. Although Linda can describe parallel programs flexibly by supporting generic many-to-many communication, its flexibility becomes an obstacle to faster execution of Linda programs in distributed environments. Naive implementations of Tuple Space, in which the nonexclusive and weak protocols are unavailable, requires unnecessary message exchanges. For example, it requires them to guarantee the exclusiveness of In even when no conflicts of In's occur. In such a case, however, MTS does not guarantee it, and reduces message exchanges by using the nonexclusive protocol. Flow-analysis of a program determines the best combination of the protocols under Condition 2 in the previous section. This section shows typical examples where it is clear that they satisfy Condition 2.

The first example is Server-Clients communication. In the client-server paradigm, many clients request and get replies from a single server. The weak protocol is suitable in this case. For example, an electronic mail system uses this type of communication. A node that attempts to send mail executes Out and generates a tuple that contains a receiver's address and the contents of the mail. If you send a mail to a person 'Chiba' in the Information Science Department, the generated tuple is ("Chiba", "info science", "...message..."). This tuple is removed by a node whose domain name matches "info science" or which is a transmitter to that department. The sender node is a client and the receiver node is a server. Note that there is only one server corresponding to each domain, while there are many clients. Figure 3 shows a skeleton of Server-Clients communication. All In's included by both **server()** and **client()** can use the weak protocol without breaking the consistency. Because there is a single server, the In of the server never conflicts with another In. Because **client_id** is an identifier unique to a client, the In of a client never causes a conflict either. Tuples that no other TS operation manipulates can be removed by In according to the weak protocol.

```
/* C-Linda program */
server()
{
  int client_id;
  ...
  while(1){
    in("request", ? client_id, ? argument);
    ...
    out("reply", client_id, result);
  }
}

client()
{
  int client_id = unique identifier;
  ...
  out("request", client_id, argument);
  in("reply", client_id, ? result);
  ...
}
```

Figure 3: Server-Clients Communication

```
/* C-Linda program */
server()
{
  int index = 1;
  ...
  out("token", 1);
  while(1){
    in("request", ? index, ? argument);
    ...
    out("reply", index, result);
    ++index;
  }
}

client()
{
  int index;
  ...
  in("token", ? index);
  out("token", index + 1);
  ...
  out("request", index, argument);
  in("reply", index, ? result);
  ...
}
```

Figure 4: Merged Stream

This example assumes that a server picks requests at random. When a server reads a stream from many clients (merged stream), the weak protocol cannot applied to In. Figure 4 is a skeleton of this case. The stream consists of a series of numbered tuples. The second field "index" of a tuple is an index number. To get successively incrementing index number clients read and update a tuple whose first field is "**token**". This tuple represents the end of the stream. If many clients attempt to get an index number simultaneously, it must

```
/* C-Linda program */
P1()
{
  int i;

  out("X", 1);
  while(1){
    in("X", ? i); out("X", i + 1);
  }
}

P2()
{
  int x, y;

  out("Y", 1);
  while(1){
    read("Y", ? y); read("X", ? x);
    if(x > y){
      in("Y", ? y); out("Y", x);
    }
  }
}

P3()
{
  int x, y;

  while(1){
    read("Y", ? y); read("X", ? x);
    assert(x >= y);
  }
}
```

Figure 5: Shared Variables

be read and updated correctly. The `In` that removes this tuple, hence, cannot use the weak protocol but the strict protocol because it must have access to the index.

The last example, in Figure 5, shows a case using the nonexclusive protocol. `In` may use the nonexclusive protocol instead of the weak protocol when other `Read`'s read the removed tuple. Figure 5 shows an example, which was originally presented in [1] (the original example did not use Tuple Space). The tuples `("X",` *number*`)` and `("Y",` *number*`)` are used as shared variables `X` and `Y`, whose values are *number*. Process `P1` increases (the value of) `X`, and process `P2` attempts to keep `Y` equal to `X`. Process `P3` verifies that `X` is greater than or equal to `Y`. This verification always succeeds because `P3` reads `Y` before `X`. If the `In` of process `P1` uses the weak protocol, an inconsistency may occur. Suppose that `P1` removes a tuple `("X", 3)` and generates a tuple `("X", 4)`. Process `P2` reads the tuple `("X", 4)`, removes a tuple `("Y", 3)`, and generates a tuple `("Y", 4)`. The 'out' message from `P2`, which represents generation of the tuple `("Y", 4)`, may arrive at the node containing process `P3` before the 'erase' message, which represents removal of the tuple `("X", 3)`, arrives from `P1`. If process `P3` reads the tuples `("Y", 4)` and `("X", 3)`, then the assertion of that process fails. Figure 6 represents the history in this situation. If the nonexclusive (or the strict protocol) is used, the assertion never fails because process `P1` does not generate `("X", 4)` until messages representing the removal of `("X", 3)` arrive at all nodes.

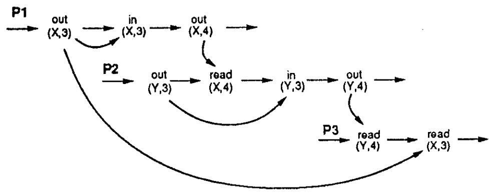

Figure 6: History

5 Experiment

A prototype of MTS is implemented on Sun SPARC workstations (SunOS 4.0) connected with Ethernet. Processes communicate with each other through the UNIX sockets with TCP/IP. A socket is not connected for every message sending but is connected once before a program execution begins. Two processes, such as a Tuple Space server (called *TS server* below) and a client process, run on each host. They communicate with each other also through a socket. A program using Tuple Space runs on a client process. When a TS server receives, from a client process, a request for carrying out a TS operation, the server executes it. The server maintains a replication on its host and other hosts according to protocols. A protocol for executing `In` is specified by a client process.

This experiment uses a fixed-size (44-byte) tuple whose elements are integers, and it measures the time during which a tuple is passed from one client process to another. The result of this measurement is compared with the time during which the same amount of data is passed directly through a socket instead of through Tuple Space. Because a socket internally buffers passed data, the experiment uses the following "ping-pong" program, in which two processes exchange tuples 100 times, to decrease the influence of buffering. Furthermore, to force immediate delivery of data, an option `TCP_NODELAY` of TCP is set [12].

```
process Ping;
begin
  for i = 1 to 100 do begin
```

```
    Out("ping", i);
    In("pong", i);
  end;
end;

process Pong;
begin
  for i = 1 to 100 do begin
    In("ping", i);
    Out("pong", i);
  end
end;
```

Table 1 lists the transfer time for one data passing, in which one Out and one In are executed. The transfer time is measured for three cases where In's use either the strict, nonexclusive, or weak protocol respectively, and for a case of the UNIX socket. For the case of the UNIX socket, transfer time is the time during which data is sent by one client process and received by another. The time was measured at night, when several other user processes run. Ping and Pong processes run on different hosts, and there is no other client process that uses Tuple Space. The weak protocol performs significantly faster than the other two protocols. The nonexclusive protocol is not much faster than the strict protocol. This is probably because the TS server that sends 'remove' messages in the strict protocol does not wait until the messages are received but continues to execute. Although passing data with a socket is much faster than with the weak protocol, the rates are comparable considering that data is passed through two TS servers in the weak protocol.

Table 1: Transfer time of one data passing (in msec)

# of replications	2	3	5
strict	9.0	9.9	14.5
nonexclusive	8.3	8.9	13.1
weak	5.5	6.0	7.8
socket	1.5		

6 Conclusions and Future Work

Tuple Space is suitable for communication facilities in distributed systems. It enables flexible description of many types of communication and data-sharing. It has the advantage that it can be implemented with relaxed consistency. This paper presents MTS, which is an implementation that exploits this advantage. It permits efficient execution of Linda programs in a distributed environment. In our experiment [5], MTS improves execution of Linda programs by up to about 40%. MTS provides for TS operations several protocols that guarantee different degrees of weakened consistency. Using the protocols together optimizes execution of Linda programs. Although some protocols guarantee weaker consistency than relaxed consistency, MTS provides relaxed consistency from the viewpoint of a programmer. For this purpose, the protocols must be selected for each TS operation under the conditions described in section 3.

There are several research activities of replications maintained with weak consistency [10, 16]. They exploit application semantics to make consistency weaker. They maintain their replication with coarse-grained optimization, while MTS maintains with fine-grained optimization. ISIS [3] is a system whose communication primitive is message broadcasting, and provides different degrees of weakened broadcast protocols. In this system, a programmer selects appropriate protocols and broadcasts messages accordingly. This system optimizes message broadcasting with application semantics.

We plan to develop a compiler that analyzes a Linda program and assigns an appropriate protocol for each TS operation. In our prototype MTS, a programmer must select a protocol for each In. Complete fine-grained optimization requires flow-analysis by a compiler.

7 Acknowledgments

We thank Satoshi Matsuoka for his valuable suggestions that helped us to clarify and organize our work. We also thank the members of the Masuda Research Group for their helpful comments and discussion during the course of this work.

References

[1] Bal, H. E. and A. S. Tanenbaum, "Distributed Programming with Shared Data," in *Proc. of the International Conference of Computer Languages*, pp. 82-91, 1988.

[2] Bennett, J. K., J. B. Carter, and W. Zwaenepoel, "Munin: Distributed Shared Memory Based on Type-Specific Memory Coherence," *ACM SIGPLAN Notices*, vol. 25, no. 3, pp. 168-176, 1990.

[3] Birman, K. P. and T. A. Joseph, "Reliable Communication in the Presence of Failures," *ACM Trans. Comp. Syst.*, vol. 5, no. 1, pp. 47-76, 1987.

[4] Carriero, N. and D. Gelernter, "Linda In Context," *Commun. of the ACM*, vol. 32, no. 4, pp. 444–458, 1989.

[5] Chiba, S., K. Kato, and T. Masuda, "Optimization of Distributed Communication in Multiprotocol Tuple Space," in *The Third IEEE Symposium on Parallel and Distributed Processing*, 1991.

[6] Gelernter, D., "Generative Communication in Linda," *ACM Trans. Prog. Lang. Syst.*, vol. 7, no. 1, pp. 80–112, 1985.

[7] Hutto, P. and M. Ahamad, "Slow Memory: Weakening Consistency to Enhance Concurrency in Distributed Shared Memories," in *Proc. of the 10th International Conference on Distributed Computing Systems*, pp. 302–309, 1990.

[8] Krieger, O. and M. Stumm, "An Optimistic Algorithm for Consistent Replicated Shared Data," in *Proc. of 1990 Hawaii International Conference on System Sciences*, vol. 2, pp. 367–375, CS Press, 1990.

[9] Kung, H. T. and J. T. Robinson, "On Optimistic Methods for Concurrency Control," *ACM Trans. Database Syst.*, vol. 6, no. 2, pp. 213–226, 1981.

[10] Ladin, R., B. Liskov, and L. Shrira, "Lazy Replication: Exploiting the Semantics of Distributed Services," in *Proc. of the Workshop on Management of Replicated Data*, pp. 31–34, IEEE, 1990.

[11] Lamport, L., "Time, Clocks, and the Ordering of Events in a Distributed System," *Commun. of the ACM*, vol. 21, no. 7, pp. 558–565, 1978.

[12] Leffler, S. J. et al., *The Design and Implementation of the 4.3BSD UNIX Operating System.* Addison-Wesley, 1989.

[13] Leler, W., "Linda meets Unix," *IEEE Computer*, vol. 23, no. 2, pp. 43–54, 1990.

[14] Li, K., *Shared Virtual Memory on Loosely Coupled Multiprocessors.* PhD thesis, Dept. of Computer Science, Yale Univ., 1986.

[15] Matsuoka, S. and S. Kawai, "Using Tuple Space Communication in Distributed Object-Oriented Languages," in *Proc. of Object-Oriented Programming Systems, Languages, and Applications*, pp. 276–284, 1988.

[16] Triantafillou, P. and D. Taylor, "Using Multiple Replica Classes to Improve Performance in Distributed Systems," in *Proc. of the 11th International Conference on Distributed Computing Systems*, pp. 420–428, 1991.