Communicating Process Architectures – 2003
Jan F. Broenink and Gerald H. Hilderink (Eds.)
IOS Press, 2003

Distributed Shared Memory in Global Area Networks

H. H. HAPPE and B. VINTER
Department of Mathematics and Computer Science, University of Southern Denmark

Abstract. Distributed Shared Memory (DSM) has many advantages in heterogeneous environments, such as geographically distant clusters or The Grid. These includes: locality utilization and replication transparency. The fact that processes communicate indirectly through memory rather than directly, is giving DSM these advantages.

This paper presents the design of Global PastSet (GPS) which is a DSM system targeted at global area networks. GPS is based on the DSM system PastSet [1] that has been very effective in homogeneous cluster environments. GPS utilizes consistency control migration and replication to scale in heterogeneous environments. This has resulted in a token-based mutual exclusion algorithm that considers locality and an algorithm for locating replicas. GPS has been simulated in multi-cluster environments with up to 2048 nodes with very promising results.

1 Introduction

Message passing systems (e.g. the Message Passing Interface[1] – MPI) have been very successful in homogeneous cluster environments because of good performance and the rather simple programming model. The good performance stems from the fact that the developer explicitly states which processes that should communicate. This means that if the developer has done it right, there will be no unnecessary communication.

When moving to more heterogeneous and dynamic environments message passing applications must be custom made to address the nature of a specific environment. Also, message passing systems rely on node availability and in case of errors the application must handle its recovery. It is not possible to make message passing systems automatically adapt in these dynamic environments because message destinations are explicitly stated and are part of the application's logic.

In DSM systems processes do not communicate directly with each other, but indirectly through memory. This makes it possible for a DSM system to solve several problems transparently.

- **Replication**: The indirect communication makes transparent replication possible because data is addressable.
- **Fault-tolerance**: By using active replication to maintain some level of redundancy transparent fault-tolerance can be achieved.
- **Locality**: Because the reading processes do not have to be known at the time of writing it is possible to prioritize local read requests higher than remote. For example, if many processes were competing to read from a distributed bounded buffer, those processes that are close to the actual location of the next entry could be served first. This makes

[1]http://www-unix.mcs.anl.gov/mpi

the system unfair and introduces the possibility of starvation, but still might increase performance considerably for some applications. Also, it is possible to make a DSM system fair and without starvation, if needed.

- **Focus on functionality**: The developer can focus on the functionality of an application instead of the environment that it has to be deployed in. This is not much of a problem in homogeneous environments where every process can be viewed as equal and communication between processes has the same cost (if each node only runs one process), but in heterogeneous environments such as the Grid this becomes very complex. Also, heterogeneous environments could dynamically change over time, which would be very hard for a developer to address using message passing. DSM systems can adapt in such a dynamic environment because the choice of destination process can be postponed until there is a requesting one. This can be the responsibility of the system so that the developer can concentrate on functionality.

Most research in DSM has been targeted at homogeneous clusters, but some work has gone into making DSM work in a global context. Unify [2] is a *shared virtual memory* system made to scale beyond clusters. Apart from accessing the virtual address space as conventional *random access memory*, it can also be accessed as *sequential access memory* and *associative access memory*. These other types of memory access makes it possible to weaken consistency. Globe [3] is a distributed object system where method invocation can be done as a remote procedure call or by migrating the target object to the calling node. This, together with various consistency models, can be utilized to make Globe scale in a global context. In Global Arrays [4, 5] an extra level of remote memory is introduced to make it scale in multiple distant cluster environments. From a node's viewpoint this gives three levels of memory; local memory, remote memory at a node in the local cluster and remote memory at a node in a remote cluster.

2 PastSet

In PastSet [1], interprocess communication (IPC) is done through what is called an *element*. Basically an element is an infinite virtual array of indexed data-blocks that can vary in size (see Fig. 1). A data-block is denoted a *tuple*, because of PastSet's resemblance to systems based on an abstract tuple space. In the original version of PastSet, tuples had a fixed size for each element, but this is not required.

2.1 The Element

The way one reads and writes an element is like a bounded FIFO-buffer, but it is also possible to read a specific index in the array. There are three variables that control the state of the bounded buffer. *first* points to the index first written but not yet read, and *last* points to the index that should be written next. *delta* is a value that defines the size of the buffer and thus the maximum distance between *first* and *last*. Tuples with an index that is lower than the *first* pointer is not necessarily disposable, it depends on the element's ”garbage” policy.

2.2 Operations

PastSet preserves a sequential consistency among all operations on an element. This makes it very user friendly for the developer. There are three main operations that a process can invoke on an element.

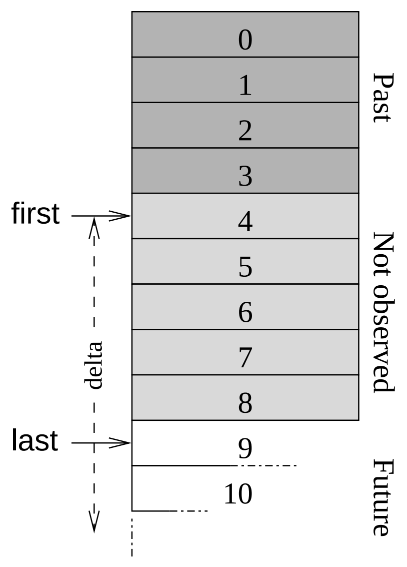

Figure 1: The PastSet 'element'.

- `move(tuple)` writes a tuple to the element at the index referenced by the *last* pointer. The *last* pointer is then incremented by one. If the distance ($last - first$) between the *first* and *last* pointer is equal to the *delta* value the operation blocks until the distance becomes smaller.
- `observe` reads the tuple referenced by the *first* pointer and the pointer is then incremented by one. If the *first* pointer is equal to the *last* pointer the operation blocks until the *last* pointer becomes greater than the *first* pointer.
- `observe(index)` reads the tuple referenced by the index. If the index has not been written yet, the operation blocks until it becomes available. This operation does not change the *first* and *last* pointer.

Tuples are immutable when first written because no operation makes it possible to update an existing tuple.

3 Global PastSet

This section describes the design of Global PastSet (GPS). To see if DSM makes sense in a global context performance is the objective of this design.

In the original version of PastSet described in [1], an element where located and controlled at one node. Therefore, all communication with an element had to go through this single node. This simple approach worked very well in homogeneous clusters, but the higher latencies imposed by a global environment makes it inefficient.

To avoid the impact of high latencies, GPS distributes elements to the involved nodes. This is done by changing the way an element is controlled and stored:

- **Element control migration**: When a node executes an operation on an element the control of the element is migrated to this node, which then can carry out the operation locally. Also, the node could execute multiple operations while it has the control. This way many operations can be executed at one location (e.g. a cluster) before control is migrated to a remote location.

- **Replication**: When a node receives or writes a tuple it stores a local copy of the tuple. This way the tuples are scattered across all involved nodes and get implicitly replicated when a node reads a tuple. Active replication could also be introduced to address availability and fault-tolerance issues. This excessive form of replication gives no coherence problems because tuples are immutable. The only problem that needs to be addressed, is that of locating the nearest replica.

To gain control of an element a node has to have exclusive access to the state of the element. The state of an element consists of the *first* and *last* pointers together with the *delta* value. A custom made distributed mutual exclusion algorithm has been developed to address this problem. This algorithm utilizes locality by favoring local requests and hereby minimizing the traffic through high latency lines.

3.1 Mutual Exclusion

The algorithm is token-based, meaning that the node holding the token has exclusive access to an element's state. Because an element's state consists of a rather small amount of bytes it is stored in the token. This way an element's state is always available at the node that has exclusive access to it.

The nodes that are using an element are part of a tree topology where the root node holds the token (Fig. 2.a). Each node has a reference to a parent node except the root node. This ensures that there exists a path from each node to the root node and therefore to the token. Also, it makes it very easy to join the group of nodes that are using an element by finding one of the others through some kind of naming service.

When a node executes an operation that needs to update the element's state it most get hold of the token. If it does not already have it, it sends a request to its parent.

When a node receives a token request it forwards the request to its own parent, unless it has a pending request itself. In the latter case the token will eventually arrive at the node and the request is therefore added to a local queue. If the receiving node holds the token the request is added to a queue in the token.

After the root node is done updating the element's state the token must be passed on to the next waiting node, if any. This is done by finding the nearest node in the token request queue and send the token to this new root node. The old root makes the new root its parent. This implies that every node must have a way to decide the cost of communicating with other nodes (e.g. IP addresses, geographical distance, performance monitoring). This way of passing the token shortens the path that the token has to travel to reach all requesting nodes.

When the token arrives at a node, this node's local queue is added to the token queue. Then the pending operations allocates indexes to observes and/or moves, after which the token can be passed on.

Allocating the needed indexes take little time and therefore if there are many pending requests in the system the token will move from node to node rather quickly. The result can be that new requests are chasing the token, which creates a long tail of requests behind the token. Forwarding these requests impose a rather big load on the involved nodes and also on the network. In order to avoid this, a node records the source of the last request it has forwarded. Now when it receives another request it can forward it to this *last seen* node. This node either has a pending request for the token or may already have seen the token in which case the request is just forwarded as described above. If the *last seen* node has not seen the token when the request arrives a distributed queue of requesting nodes will be build in front of the token path. Otherwise, the request will take a shortcut to the token. Though this is very effective, it introduces a locality problem.

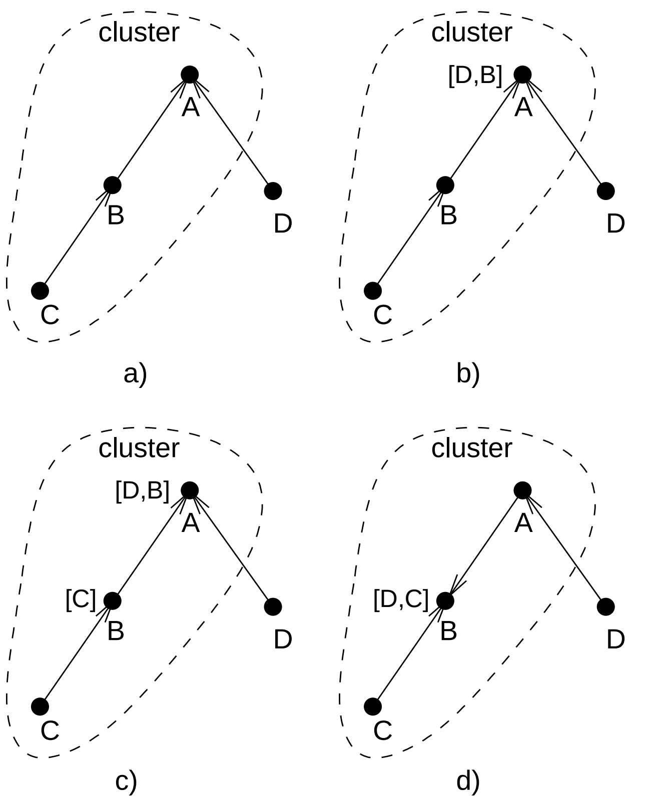

Figure 2: Tree topology. The root holds the token.

In most cases a node will have a local community (LAN, cluster, etc.) and requests from this community should not be forwarded to nodes on the outside, unless it is absolutely necessary. While a node does not know its local community, the cost of communicating with the *last seen* node defines a community in this context. In Fig. 3.a node D (*last seen*) defines node B's local community, by including all those nodes with a lower cost of communication. Now all requests coming from nodes with a higher or equal cost of communication are forwarded to D, because D's request is likely to be served before. Requests from nodes with an equal cost of communication are also forwarded to D because those are likely to be part of D's real local community. In Fig. 3.b C is inside B's community and therefore the request is forwarded to the parent. This is because r_C should not be added to a remote queue (D might be located on the other side of the world). It should rather exploit the locality in the token tree.

Fig. 3.c shows a scenario that could break locality. If C and A are local to each other and remote to the rest, the token might take a detour because r_C is queued at D. This will not always be true, because it is not certain that r_C can catch up with D's request if r_C is just forwarded to B's parent. It would have to be able to do this in order for A to make the right locality decision, when passing on the token. On the other hand, if local communities share the same subtree in the token tree, this would not happen.

3.1.1 Example

Fig. 2 shows an example of how the algorithm works. Fig. 2.a shows the initial configuration of the tree. Nodes A, B and C are part of a cluster and node D is remote to this cluster.

In Fig. 2.b A has received a request first from D and then from B, which both have been added to the token request queue.

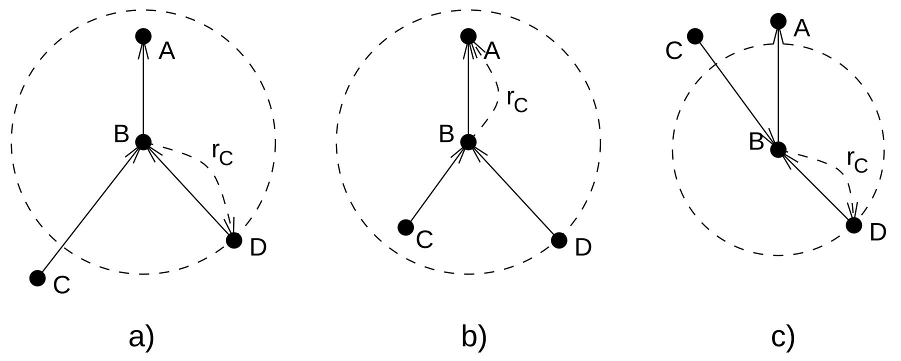

Figure 3: Request forwarding. The dashed circle defines B's local community bounded by the cost of B communicating with D - the *last seen* node indicated by the dashed arrow. a) The request r_C is forwarded to D because C is outside B's local community. b) The request r_C is forwarded to A because C is inside B's local community. c) A scenario that could send the token on a detour, if C and the node that holds the token (A) are local to each other.

In Fig. 2.c C has send a request to its parent B which has added the request to its local queue, because it already has a pending request.

In Fig. 2.d A has passed the token to B even though D's request arrived first at A. This is because B is closer to A than D. At B the local queue has been added to the token queue.

3.1.2 Starvation and Fairness

In most scientific applications starvation and fairness in IPC are not an issue because the main priority is to keep the nodes calculating. If CPU utilization is high all nodes will eventually be served. On the other hand if an application has a high communication overhead, it might as well be carried out by nodes local to each other if possible.

The mutual exclusion algorithm introduced above is not fair and starvation could occur. Fairness could be introduced by removing the utilization of locality and starvation could be avoided by forcing a request to be served if it has been postponed too long (with a counter or some kind of timeout).

3.2 Move and Observe

This section describes how the custom made mutual exclusion algorithm is utilized to implement the `move` and `observe` operations.

The `move` and `observe` operations can only be carried out when the conditions $last - first < delta$ and $first < last$ holds respectively. Therefore, should the token only be passed on to a node that can complete a pending operation (the condition holds).

To distinguish between moves and observes a node explicitly requests a move or an observe, instead of just requesting the token. Nodes also distinguish between pending moves and pending observes when queuing requests locally to avoid dead-locks. A node must also distinguish between move and observe requests when saving a reference to the source of the last seen request.

The fact that the request states which operation to execute also makes it possible for other nodes holding the token to carry out the operation on behalf of the requesting node. Therefore, it is possible to constrain the token to certain nodes for security or dependability reasons.

With these additions the operations can now be specified:

index	references	local
3		true
4	n_2, n_5	false
6	n_4, n_9, n_{12}	true
10	n_4, n_{10}	false

Table 1: Tuple table example.

- `move(tuple)`: The issuing node gets hold of the token. Then it inserts the tuple with *last* as index, given that the move condition holds, and increments *last* by one.

- `observe`: The issuing node gets hold of the token. Then it records *first* as the tuple index to read, given that the observe condition holds, and increments *first* by one.

When the root node is ready to pass on the token, it is passed to the closest node in the token request queue that has a pending request which can be carried out (the condition holds). Also, observes are served before moves if their sources are equally close to the forwarding node. This makes it more probable that the observing node gets a reference to the node with the tuple it needs, because the distance between the *first* and *last* pointer is minimized (see section 3.3).

3.3 Locating replicas

To guarantee that a node can locate a tuple with a given index there will have to be a deterministic path from each node to each tuple. This section describes what information a node must maintain to ensure that this path exists and how this information is used to locate a tuple.

3.3.1 Tuple Table

A node should maintain a tuple table (see Tab. 1). For each index that the node has information about, there is an entry in the table. An entry holds a list of references to nodes that have the tuple and it tells whether the node has the tuple locally.

In the example there are no references to other nodes at index three, but the node has the tuple locally. For index six the node holds the tuple locally, but it also has references to other nodes that hold the tuple. This could seem redundant but it makes it possible for the node to flush the local copy while still being able to find it again.

To ensure that there exists a deterministic path from each node to each tuple a constraint is imposed on tuple tables. This constraint must ensure that a node that has a copy of a tuple with index i must also know where to find the tuple with the index $i-1$. For $i = 0$ this is obviously not required.

All that are needed to uphold this constraint is that the token holds a reference to a node holding the last written tuple. Then the next node that moves a tuple can store this reference in its tuple table.

How this constraint together with the token tree is utilized to locate tuples is explained below.

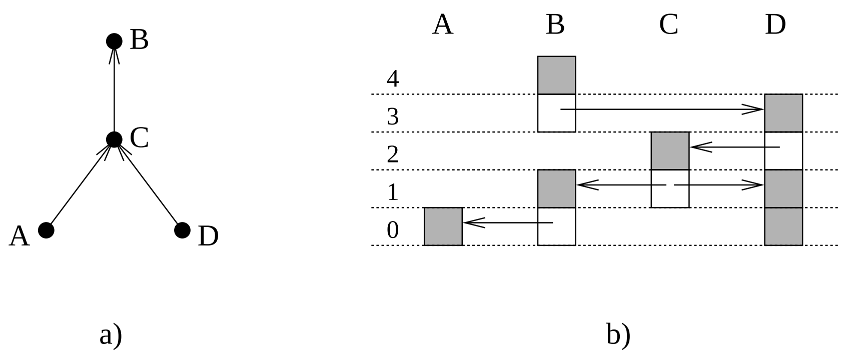

Figure 4: Tuple request paths. a) Logical tree for mutual exclusion for nodes A-D. b) Tuple tables for the four nodes. Dark squares indicate that the node has a copy of the tuple locally and an arrow indicates where it can find it. If there is no square the node does not know anything about the tuple.

3.4 Tuple back-trace

The tuple table constraint described above ensures that for each tuple with index i that a node holds locally there exists a path to the location of each tuple with index $j < i$ (see Fig. 4).

If a node needs a tuple with index i it sends a request to the node that has the tuple with index j, where j is the smallest index so that $j \geq i$. When a node receives a request for a tuple that it does not hold locally it forwards the request in the same manner.

Fig. 4.b illustrates how these paths work. If node C wants the tuple with index $i = 0$, it sends a request to node B because the tuple with index $j = 1$ is the nearest tuple, so that $i \leq j$, that C can find. The request is again forwarded from B to A following the same rule and A has a copy of the requested tuple. If B ceased to exist there would still be a path to the copy at D.

If there are multiple nodes to choose from when forwarding a request (as in the example) the nearest node is chosen.

3.5 Parent forwarding

Fig. 4.b shows that there is not necessarily a path from all nodes to all tuples, but if the logical tree for mutual exclusion is used together with tuple back-tracing, this path exists. This is easy to prove. Following the path to the tree's root a request can find the token, which has a reference to the location of the last written tuple. Now it is possible to back-trace to the required tuple. If for example node C in Fig. 4 wants the tuple with index $i = 3$ it sends a request to its parent B which starts the back-tracing to D.

In many situations the request will be able to start the tuple back-tracing on its way toward the root. This will relieve the root node and make the path shorter.

3.6 Non-existing tuples

In the case of an indexed observe of a tuple that has not been written yet, a process should block until it becomes available. Therefore, all nodes with a pending request for a non-existing tuple should receive a copy of the tuple after it is written, but how is this achieved efficiently?

A tuple request for a non-existing tuple will eventually reach the root node where it discovers that the tuple does not exist. These requests could simply be recorded in the token and be served when the tuple becomes available. When a node writes the tuple it would have to send copies to all waiting nodes listed in the token. For some applications this would result in a huge token payload caused by too many tuple requests, which would slow down token forwarding. Also, the time used for sending copies to waiting nodes would increase proportionally to the number of pending requests.

This problem is solved by parallelizing the copying so that nodes receiving the first copy, forwards it to other waiting nodes and so forth. To accomplish this a tree of waiting nodes is build for each requested non-existing index. In this tree each requesting node knows its child nodes (if any) and the root of the tree is maintained in the token. The token and each node can have an individual fixed number of children and this number defines the max number of copies a node wants to forward. This also helps distribution of existing tuples.

Now how is the tree created? When a node receives a tuple request it checks if it has already got a pending request itself, in which case it adds the requesting node as a child. If it cannot have more children it forwards the request to one of the children it already has. The choice of child to forward to is based on locality, meaning that the closest child will be chosen. If there is a choice of equally close children, requests are forwarded to these in a round-robin fashion. If it does not have a pending request it will forward the request to its parent.

The root of the distribution tree is handled a bit differently. If there is room in the token a request is just added like described above. If not, the existing requests are searched to see if one originates from a node further away from the token's current position, compared to the source of the new request. If such a request exists it is replaced with the new request and sent to the source of the new request. Otherwise the new request is forwarded to one of the nodes in the token. This will build subtrees with nodes that are local to each other, because the token is not forwarded to remote nodes unless it is absolutely necessary. For example when a distribution tree has been build in a cluster where the token exists and the token is transfered to another cluster, local requests in this other cluster will replace the requests in the token. This will have the effect that there will be few edges between the two clusters in the distribution tree.

When a requested tuple is written the writing node has the token and can therefore send copies of the tuple to the children of that index (if any). The tree ceases to exist when the tuple it represents becomes available and it is automatically removed as each node has received a copy of the tuple and forwarded it to its children.

The optimal configuration of a tree would naturally be one that exploited the network's topology to achieve a minimal tuple distribution time, but the information to do this is not available. First of all it is not known which node will write the tuple while the tree is created, secondly the set of nodes requesting the tuple is not predetermined.

3.6.1 Piggybacking

To minimize the number of hops required to locate tuples the tuple tables are filled with extra tuple information. This is accomplished by piggybacking tuple information when sending messages. This redundant information is justified by the fact that bandwidth is increasing every day, while latency improvements are rare and bounded by the speed of light. Therefore, if redundant information can lower the number of hops that requests have to take, performance should be improved considerably.

The information that is piggybacked is small tuples and references to nodes holding larger tuples. Small tuples are sent fully, because they are comparable to references in size. The type of message that piggyback information is attached to dictates in some cases the contents of this information. For example when the token is passed on to a node that needs it for an observe operation, information about tuples with indexes from *first* and up follows the token. For other messages there is no way to connect helpful information in regards to the message

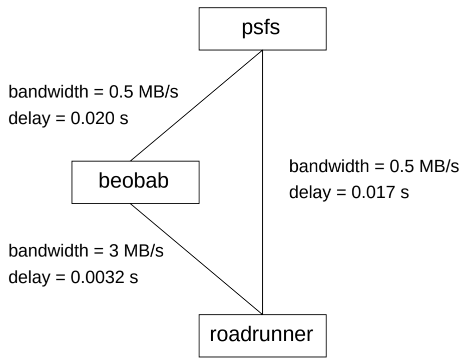

Figure 5: Simulation environment.

itself. Information about the last known tuple is always piggybacked, which helps to find other tuples as described in section 3.4.

It has not been investigated what effect, the number of small tuples and/or tuple references that are piggybacked with each message, has. More piggyback information should lower the number of hops a tuple request has to take. On the other hand, the added payload of more piggyback information would increase the load and take up storage resources.

4 Simulation Results

The performance of the design have been tested by simulating an environment of three clusters from the real world. The simulation has been implemented in the Ptolemy II [6] modeling system. The choice of simulation instead of a real implementation is based on two facts; First of all it has not been possible to get exclusive access to thousands of nodes. Secondly the heterogeneous nature of the environment makes it difficult to compare results.

4.1 The simulated environment

The clusters that the simulation has been based on are *roadrunner* and *beobab* which are located in two different cities in Denmark, together with *psfs* located in Norway.

Fig. 5 shows the three clusters. Also, the bandwidths and one-way null byte latencies (delay) between them are displayed. These numbers have been benchmarked in the real environment.

The real setup of the real clusters have not been simulated. Instead each simulated cluster consists of a number of nodes connected with one 100 Mbit/s switch that has a backplane with infinite bandwidth. The delay from node to node through this switch is 100 μs (benchmarked with a real 100 Mbit/s switch).

4.2 Test Programs and Initial Setups

The performance of GPS has been tested with a variety of communication patterns, instead of actual parallel applications. This is obviously because these applications require too many system resources to be run on a single workstation with Ptolemy. Therefore, a set of communication patterns, that are often used in parallel applications, have been chosen. These are as follows:

- **One-to-one**: This is simply communication between two nodes. This will show how fast a node can communicate with another node, which is not as trivial in DSM systems

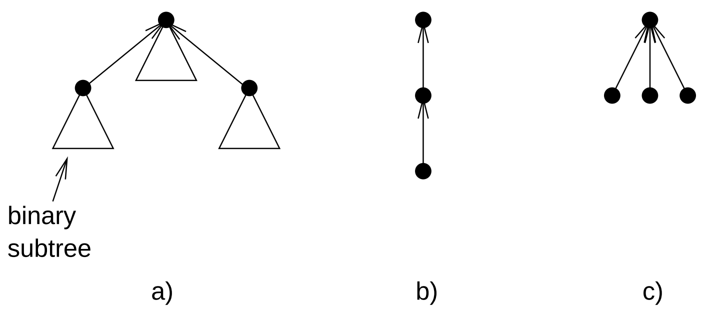

Figure 6: Initial tree configurations.

as it is in message passing systems where nodes explicitly define what node they want to communicate with.

- **One-to-many**: This resembles a broadcast from one node to all nodes.
- **Many-to-one**: All nodes sent data to one node.
- **Many-to-many**: All nodes sent data to all other nodes. This is the same as if all nodes broadcast a message.
- **Global reduction**: This is often used in parallel applications to make a global sum, minimum, maximum, etc. out of local values from each process. When the result is ready it has to be distributed to all involved nodes.

Patterns based on indexed and unindexed observes have been tested. All these patterns have been tested with different numbers of nodes and various configurations defined by these properties:

- Node configuration. Each program has been run with a various number of nodes to show how well GPS scales. One, two (*roadrunner* and *beobab*) and three cluster setups have been tested. The clusters has been given an equal amount of nodes.
- Initial token tree configuration. Fig. 6 shows the initial token tree configurations that have been tested. In the binary configuration each cluster has a binary subtree as in Fig. 6.a. In the linear configuration each node has only one child and there is only one edge between clusters (see Fig. 6.b). In the star configuration all nodes has the root as their parent (see Fig. 6.c).
- Tuple size. The programs has been tested with a tuple size of one byte and 10000 bytes.

4.3 Results

All programs with all the various configurations have been run. This amounts to hundreds of results, which would be to comprehensive to discus in this paper. The general picture is that GPS scales linearly in most runs.

The following will elaborate on some of the outstanding results that illustrates parts of the system that needs improvements.

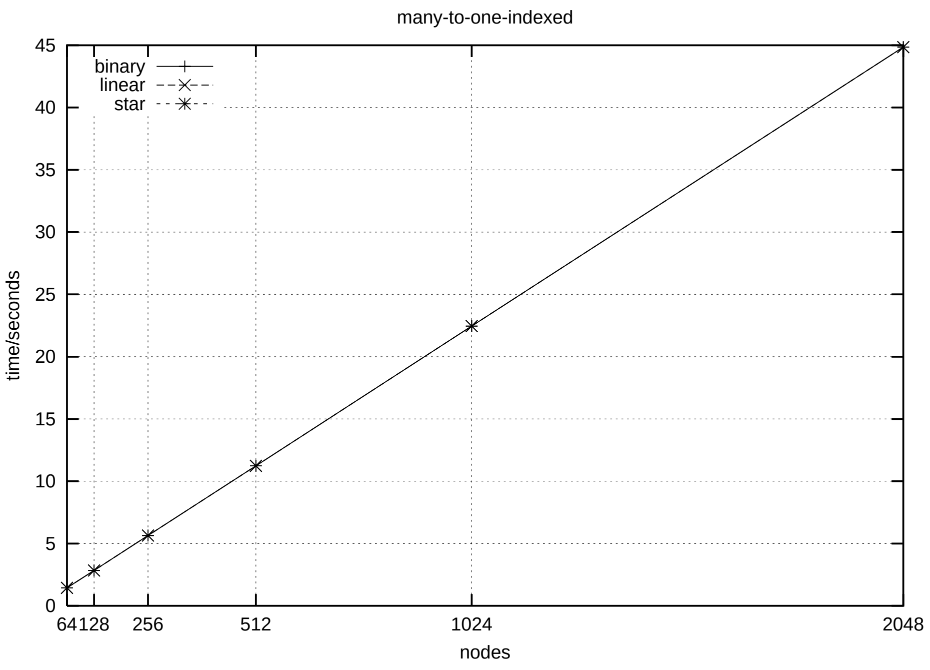

Figure 7: 3 clusters, tuple size=10000.

4.3.1 Many-to-one Indexed

In this program all nodes move a tuple, while one node in the *roadrunner* cluster makes n (the number of nodes) indexed observes. Fig. 7 shows the results for three clusters and a tuple size of 10000 bytes.

Though the results scale very well for all three tree configurations the performance is not ideal. The bandwidth between *roadrunner* and *psfs* sets a lower bound of what can be achieved. For $n = 2048$, $2048/3 = 683$ tuples must be transfered from *psfs* to *roadrunner*, which with a bandwidth of 0.5 MB/s takes 13.0 seconds. These poor results are caused by the fact that the observe operation is blocking. The extra time is spend locating tuples before they can be downloaded. This is all done sequentially because only one observe is executed at a time.

A solution to this problem would be to find many tuples in parallel. This could be done by introducing a prefetch operation that requests many tuples at a time.

4.3.2 Many-to-many Indexed

In this program all nodes move a tuple after which they make n indexed observes. Fig. 8 shows the results for three clusters and a tuple size of 10000 bytes.

The performance is very bad because many requests for a tuple arrive simultaneously at the node that has moved it. The node will send the tuple to all requesting nodes, instead of forwarding the requests to other nodes that might have or will get the tuple. This is because the design of GPS dictates that a node having a tuple locally must send it directly to a requesting node. Together with blocking observes this results in a very serial distribution of tuples.

Again a prefetch operation would improve considerably on these results, because tuple distribution trees could be build before tuples are moved. Also, a node that has a tuple transfer in progress could forward a consecutive request for the same tuple to other nodes that hold, or will hold the tuple in the future. This could be combined with wormhole routing to improve

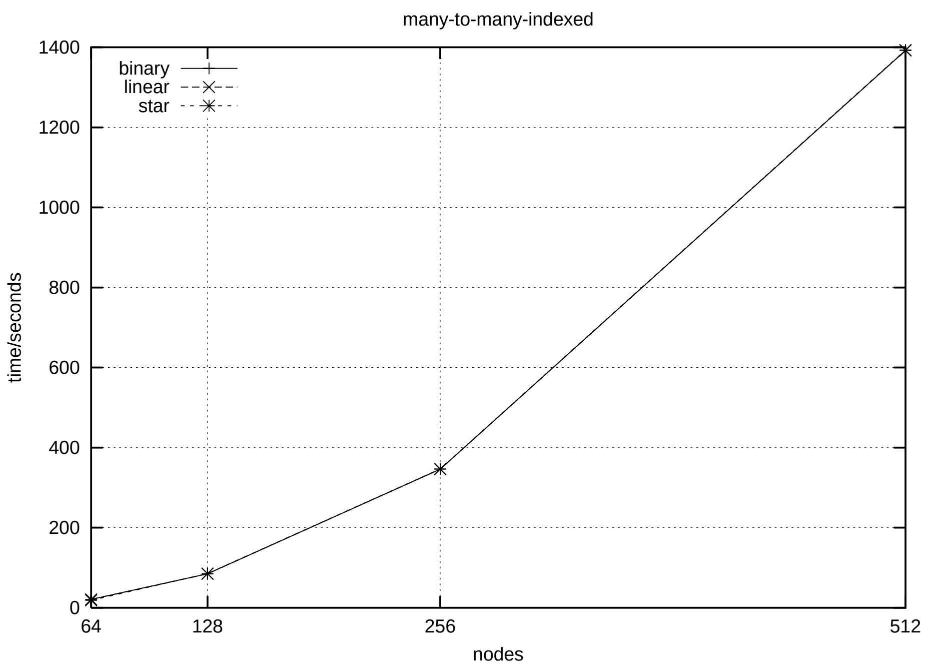

Figure 8: 3 clusters, tuple size=10000.

performance even further.

4.4 Global Reduction

To make a global reduction with GPS one node moves the first result. All other nodes makes an unindexed observe, reduces the received value with the local value and moves the result. All nodes observes the index $n - 1$ to get the result of the global reduction. Fig. 9 shows the results for three clusters and a tuple size of one byte. The results for one and two clusters look almost the same except for slightly better performance, as one would have expected.

Though a global reduction can be made to scale logarithmically, the GPS reduction will at best scale linearly. This is because the token has to visit all nodes. Having this in mind the linear and star configurations are performing ideally. The reduction is carried out locally in one cluster before it is continued in the next. For a tuple size of one byte there is a lower time limit of $2047 \times 100\mu s = 0.2047s$ ($n = 2048$), which is the minimum time for the token to travel to all nodes in a single cluster. This is close to the actual results. Also, all the results show that broadcasting the result does not add much to this time, thus it is done in parallel.

The binary configuration has a problem because the token grows very large as it travels from node to node. Each parent receives one observe request from each child which they queue locally because they have a pending request of their own. When the token arrives at a parent, one observe request is removed from the token, but two are added from the local queue. As a result the token costs more and more to forward until it arrives at the leaf nodes where no new requests are added.

This problem is not easily solved, but a solution could be to set an upper limit on how many requests the token can queue. When the queue is full, remaining requests should simply follow behind the token. The problem with this solution is that the set of nodes to choose from, when picking the next token destination, is smaller.

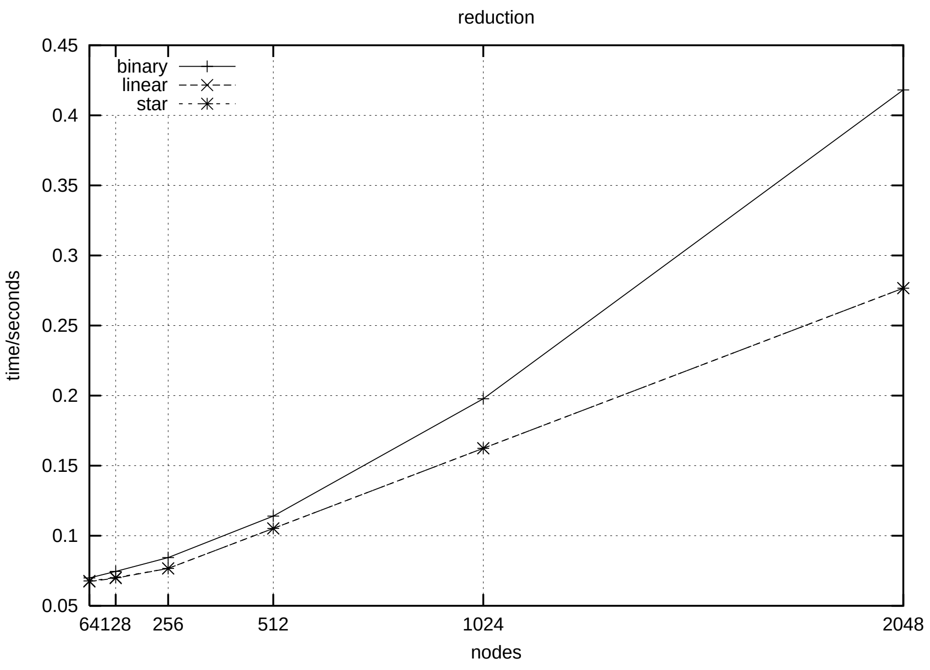

Figure 9: 3 clusters, tuple size=1.

5 Conclusion

This paper has described the design of a DSM system for global environments. The system is based on PastSet [1], a DSM system for cluster environments. The design has been tested in a simulated environment of multiple geographically distant clusters, with up to thousands of nodes.

The design has shown that there are two main problems that need to be addressed, when moving PastSet to this new environment; Some means of upholding consistency and an algorithm for locating near replicas.

The test results have shown that GPS scales linearly, with some exceptions. The problems of these exceptions have been identified and solutions to address them have been suggested.

References

[1] Brian Vinter. *PastSet: A Structured Distributed Shared Memory System*. PhD thesis, Department of Computer Science, Faculty of Science, University of Troms, Norway, 1999.

[2] James Griffioen, Rajendra Yavatkar, and Raphael Finkel. Unify: A scalable, loosely-coupled, distributed shared memory multicomputer.

[3] Philip Homburg. *The Architecture of a Worldwide Distributed System*. PhD thesis, 2001.

[4] Jarek Nieplocha, Robert J. Harrison, and Richard J. Littlefield. Global arrays: a portable programming model for distributed memory computers. In *Supercomputing*, pages 340–349, 1994.

[5] J. Nieplocha and R. J. Harrison. Shared memory NUMA programming on I-WAY. In *Proc. of the Fifth IEEE Int'l Symp. on High Performance Distributed Computing (HPDC-5)*, 1996.

[6] *Ptolemy*. http://ptolemy.eecs.berkeley.edu.