Efficient Distribution of Immutable Data Structures in the Distributed SubSystem (DSS) Middleware Library
Per Sahlin

Information Technology Computing Science Department Uppsala University Box 311 SE-751 05 Uppsala Sweden

This work has been carried out at SICS Uppsala Science Park SE-751 83 UPPSALA Sweden

Supervisors: Erik Klintskog Examiner: Richard Carlsson

Passed:
Abstract

Efficient distribution of data is the foundation in a distributed system. We have conducted a short survey and found that numerous systems present solutions for handling stateful (mutable) data structures, while stateless (immutable) data structures are often disregarded. We believe efficient distribution of immutable data structures to be important in order to realize efficient and scalable distributed applications. We present a taxonomy of immutable data structures based on their size, structure and usage. A dynamic distribution model is presented by using an optimal strategy when distributing the different categories. These distribution strategies were implemented as part of the Distributed SubSystem (DSS), a state-of-the-art middleware library.
## Contents

1 Introduction ........................................................................... 4  
1.1 Outline .............................................................................. 4  

2 Memory Distribution Models ................................................. 5  
2.1 Unstructured Page Base Memory Model .............................. 5  
2.2 Structured Memory Model .................................................. 7  
2.3 Summary .............................................................................. 9  

3 The Distributed SubSystem .................................................. 10  
3.1 The Distribution Structure ................................................ 10  
3.2 Global Naming in the DSS .................................................. 11  

4 Efficient Distribution of Immutable Data Structures ............... 12  
4.1 A Taxonomy of Immutable Data Structures ......................... 13  
4.2 Tools for Efficient Distribution of Immutable Data Structures 15  
4.3 Strategies for Distribution of Immutable Data Structures .... 16  

5 Implementation .................................................................... 18  
5.1 Protocols ........................................................................... 18  
5.2 Named Data ........................................................................ 22  
5.3 Package Format ................................................................... 24  

6 The StringVec Data Structure ............................................... 26  
6.1 Distribution ........................................................................ 28  

7 Evaluation ............................................................................. 31  
7.1 Evaluation setup ................................................................... 32  
7.2 Results .................................................................................. 33  

8 Conclusion ............................................................................ 35  

9 Acknowledgments .................................................................. 35  

A The evaluation program ....................................................... 38  
A.1 The Host node ...................................................................... 38  
A.2 The Client node .................................................................... 41
1 Introduction

Shared state distribution provides an easy to use model for distribution of data. This model makes a collection of independent computers appear to the user as a single system.

A Distributed Shared Memory (DSM) includes all distribution models that share state. There are two approaches for distributing data in a DSM system, using either an unstructured or structured memory model. The unstructured memory model distributes data in the terms of memory pages on a low level. The structured memory model distributes on a higher level and in terms of language entities.

The focus in the distribution system has mainly been to efficiently distribute and update stateful (mutable) data. Stateless (immutable) data is often disregarded and handled implicitly by being replicated. Systems using an unstructured memory model are not concerned with what kind of data is on the distributed memory pages, making it hard to efficiently handle immutable data separately. Some of the systems using a structured memory model [1, 2] allow the user to specify which data structures are immutable. But none of the systems using a structured memory model present any other distribution strategy than to replicate the immutable data structures among the nodes.

Replicating immutable data structures makes them available at multiple nodes in parallel, but this generates a possible unbound overhead, since duplicate copies can be replicated to the same node and data can be replicated without being used.

We conclude that there is a need for a dynamic solution, since there is no single optimal distribution strategy for immutable data structures. In this thesis we present a framework for efficient handling of immutable data structures. Our first contribution is a taxonomy of immutable data structures based on size, structure and usage. The second contribution is the naming of shared entities, thus making it possible to avoid both unnecessary replication and duplication of entity instances. The third contribution is introducing an immutable abstract entity structure that can be distributed as an entity representative. Thereby we avoid replicating entities to nodes where they are not used. The Distributed SubSystem (DSS) [3] is extended with an implementation of this solution.

1.1 Outline

In Section 2 we present a short survey of different Distributed Shared Memory (DSM) systems that manage shared data structures, with focus on how
immutable data structures are distributed. DSS, which is the DSM system we have used, is presented in Section 3. In Section 4 we suggest how immutable data structures should be categorized and handled. The implementation of the different strategies in DSS is described in Section 5. In Section 6 we demonstrate an example data structure implementation. The performance is evaluated in Section 7, and in Section 8 we present our conclusions.

2 Memory Distribution Models

Software based Distributed Shared Memory (DSM) models provide the user of a distributed system with a comprehensive memory model. A DSM model handles much of the complexity regarding memory distribution, data consistency, etc., and hides this from the user. A DSM system uses a DSM model and guarantees some level of data consistency. A DSM system migrates or replicates data among the participating nodes to provide the abstraction of shared memory. These systems can sometimes be provided with information about what kind of data that is shared and thereby handle it more efficiently.

We have looked at a number of state-of-the-art DSM systems and analyzed how they handle immutable data structures.

2.1 Unstructured Page Base Memory Model

One way of distributing data in a DSM system is to share memory pages among the nodes. This will give the nodes a notion of a shared memory and is a language independent memory model. These page-based systems have an unstructured memory that appears as a linear array of bytes. For efficiency reasons, it is often interesting to use a fine-grained sharing, i.e., smaller page size, and/or a more relaxed consistency protocol [4].

Coarse-grained Memory Model

Shared Virtual Memory (SVM) is a coarse-grained implementation approach, using the same techniques as local virtual memory, but sharing the memory pages among the different nodes. The IVY [5] system is a classic example implementing a SVM, a virtual memory for loosely coupled processors.

When the memory pages are updated the memory model has to ensure some level of data consistency. Sequential consistency [6] is commonly implemented using a single writer and multiple readers. This will allow only one node at the time to update the memory page, but many nodes may have read-only copies of it.

5
Each memory page is initially placed at the node that has the first access to it. The locations of the memory pages are stored in a distributed location table that is cached locally at each node. The node may have either read-access or write-access on a page, allowing the page to be distributed as a read-only copy. The read-only copies are replicated to nodes that frequently have read-access to them. If a node requests write-access, the page is located using the location table and migrated to the calling node, updating the location table. The read-only copies are either updated directly or invalidated. When different nodes simultaneously update unrelated data located on the same memory page, that memory page will be migrated back and forth between the nodes. Since the data is unrelated, the different nodes will never try to change the same part of the memory page. This so-called false sharing is a potentially serious performance problem.

In TreadMarks [7], false sharing is solved by using multiple writers. The memory page is replicated on request, allowing multiple instances of a page to coexist. On update a copy of the page, a twin, is created. After updating, the page is compared to its twin and the diff is sent to all the copies of the page, i.e., the whole page is never resent. To increase performance further, TreadMarks uses a more relaxed consistency protocol, called release consistency.

TreadMarks does not explicitly handle immutable data structures, but because of the multiple writers protocol it allows pages containing only immutable data to be replicated.

The SVM system has no knowledge about what kind of data structures that are placed on the memory pages. Mixing immutable and mutable data structures on the same memory page is very wasteful, since every update on a mutable data structure results in that all copies of that memory page are invalidated, i.e., the page has to be re-distributed. Providing the SVM system with information about the data structures, it can place mutable and immutable data structures on separate memory pages. Memory pages containing immutable data structures can then be replicated.

In TreadMarks multiple writers solve the problem with memory pages containing a mix of mutable and immutable data structures.

Systems that share memory pages give the nodes a notion of a shared memory. This is a straightforward model but since it is a low-level distribution model it disregards the structure of the data that is shared, i.e., it can not distinguish between sharing mutable or immutable data structures. As mentioned above, by providing the system with structural information, mutable and immutable data structures can be separated. But the whole memory page still has to be replicated.
Fine-grained Memory Model

Decreasing the size of each memory page, using a finer granularity, and also a more relaxed consistency protocol can potentially increase performance [4], but there is not one solution for all applications [8]. Coarse-grained systems are highly sensitive to fine-grained synchronization, i.e., sharing small data structures. On the other hand, if the granularity is too fine, large data structures will be located on many memory pages. This fragmentation results in many page misses when the data structure is accessed remotely. Shasta [9] is an example of a system that has support for variable granularity memory models. It may thereby be customized for a specific system by adjusting the memory page size. Each shared page has a home node that keeps track of each copy. Pages may be freely distributed for reading among other nodes and when a page is updated all the copies are invalidated.

By making the system separate immutable and mutable data structures, the memory pages that solely contain immutable data structures may be replicated.

2.2 Structured Memory Model

A more high-level approach to share data is to provide a special shared entity in which the user can place the shared data. The distribution will only share data that is specified to be shared. Sharing data on a data structure level makes it possible to distribute different data structures using different strategies.

To efficiently distribute data structures, they must be well structured. Well structured includes separation of unrelated data into different structures, and not having too big data structures. A possible drawback can be the overhead created when distributing large amounts of small data structures, since each data structure need its own distribution structure.

Object-based Memory Model

Object-based distribution of data, using a structured memory model, offers a fine-grained memory distribution. On a remote invocation of a global object its state is either replicated or migrated \(^1\). Sequential consistency is provided for migrating objects, since there is only one instance of each object. The

\(^1\)In the Emerald system [1] objects can contain processes, so a remote invocation can be both that the data is migrated to a remote process and that the process is migrated to the remote data.
user can specify what data structures are shared and to some extent how they are shared.

Emerald has a concrete type object that is immutable and can be replicated. The implementation of an object is stored in a concrete type object. When a node receives an object whose concrete type object is not present, the node requests the implementation and then links it to the node’s kernel. Other kinds of immutable data structures are not explicitly handled.

In the Orca system [10] all objects\(^2\) are replicated to all nodes upon creation. Keeping replicated objects up to date can be done by either invalidating or updating the copies. The solution relies on an underlying layer that provides reliable broadcast. Immutable objects would be treated as other objects and would be replicated to all nodes on creation.

In [11] Bal et al. evaluate the use of two different protocols for replicating objects in Orca. The invalidation protocol uses a partial replication strategy, creating a copy at a remote node using a point-to-point communication scheme and invalidates all copies on update. The update protocol uses reliable multicast for replicating the object to all nodes and to distribute updates. Handling immutable objects, the invalidation protocol would create an unnecessary structure for keeping track of the copies. The update protocol would replicate immutable objects to all nodes.

**Object Oriented Memory Model**

In the object oriented memory model a programming language is extended with distribution support for the data structures present in the language. Here we present language dependent systems and in Section 3 we present the DSS that is language independent.

These systems [12, 2, 13] are distribution extensions to existing programming languages, and to be precise they all extend Java. Sharing is often provided on a level of inheritance and only objects that have inherited from distribution object can be distributed.

The cJVM [13] is a cluster enabled implementation of the Java Virtual Machine (JVM) which provides a single system image for the Java application. Multi-threaded Java programs can be distributed without alterations. The objects are located at the node where they were created and are distributed by creating a proxy structure at the remote node. The proxy can be implemented in different ways, but the simple proxy implements Java RMI [14], i.e., all object operations are executed on the object where it is present and the results are returned.

\(^2\)Note that the Orca system is object-based rather then object-oriented; it does not support inheritance, dynamic binding, and there is non-object data in the system.
JavaParty [2] provides a distribution layer on top of the JVM. It allows objects to be declared as remote and will then take care of the Java RMI calls for the object. In the system the user can add proxy protocols, customizing how the objects should be replicated. The user can define immutable objects, but there are no special protocols for immutable data structures.

The third system we looked at is the Manta [12] system. The user here defines a group of objects to be replicated as a cluster. The compiler and the runtime system together determine which methods are read-only and these are always executed locally. Replicated objects have some restrictions so any Java object can not be distributed. Immutable objects would have read-only methods and would thereby be distributed freely.

By adding distribution on the language level, the distribution system can distribute different data structures in different ways. The system can take advantage of its knowledge about the characteristics of a data structure to efficiently distribute the structure. It is crucial for efficiency that the data structure is structured wisely, i.e., separating unrelated data in different structures, etc.

2.3 Summary

The DSM systems included in our survey have all focused on distributing mutable data and optimizing this. With only few exceptions [2, 1] immutable data structures are left out of their models and are handled implicitly.

Page based systems would need to place mutable and immutable data structures on separate memory pages to be able to handle immutable data structures efficiently. Page based systems are best suited for closed distributed systems, i.e., clusters, since they depend on a reliable communication protocol.

Object based and object oriented systems have a structure that is favourable for distributing immutable data structures. But still we have only found two systems that identify the immutable data structures, the Emerald and the JavaParty systems. These systems implicitly handle immutable data structures by allowing them to be replicated, with the exception of Orca, which replicates all objects on creation.

None of the systems in our survey have presented a dynamic and efficient solution for handling immutable data structures. In the JavaParty system it is possible to declare an immutable object and to create an own proxy that handles this object, but nothing is provided.
3 The Distributed SubSystem

The DSS is a language independent middleware library intended to be coupled to a programming system (PS)\(^3\) with threads and some form of automatic memory management, i.e., garbage collection. It provides an abstract entity structure that allows for distribution of data structures. Different abstract entities capture the different distribution characteristics for different data structures. It has been used for extending the programming languages Oz [15] and C++ with distribution support. Internally the DSS has a layered structure (see Figure 1), consisting of an API to be used in the PS, a protocol layer and a messaging layer. The API provides functionality for distributing entities among nodes, with a choice of protocol. Different protocols provide different distribution methods with different levels of consistency.

The information is marshaled for transportation using the messaging layer. The layered structure with its clear interfaces makes it easy to add and change parts of the DSS. In particular it is easy to add new protocols, leaving the message passing to the messaging layer.

3.1 The Distribution Structure

The DSS provides generic distribution support for a large (potentially all) set of PS specific data structures. Most data structures share a lot of basic characteristics that can be used then they are distributed. These characteristics can be that the data structure uses read and write operations or that it is read-only.

In order to provide distributing support for a data structure, it is categorized according to its distribution characteristics, i.e., its need for distribution support. The data structure is represented by an Abstract Entity (AE) in the DSS. This AE supports the functional characteristics of the data structure. The communication between the data structure and the AE uses a mediator

\(^3\)A PS can be anything from a library to a programming language.
layer. For instance, a stateful (mutable) data structure, as an object in Java, is represented by a mutable AE that supports read and write operations. A stateless (immutable) data structure, as an atom in Lisp, is represented by an immutable AE that only supports read operations. All operations on a shared data structure will be executed through the AE.

The instances of the AEs are linked together in a coordination network, where the coordinator (Figure 2) functions as a network hub. The AE consists of three parts; a memory management strategy, a coordination strategy and a consistency strategy. These strategies are implemented as three protocols running in parallel over the coordination network.

The distribution of shared a data structure in the DSS is done by first constructing an AE, then the AE is transferred and built at the remote node. The AE defines how the data structure is handled by the strategies. This choice is made when the AE is created.

An AE is removed when it is no longer needed, for instance when the entity it represents is removed or no longer used. When there is only one AE left in a coordination network, the complete coordination network is dismounted. The data structure associated with the last AE is localized, i.e., it can be accessed locally.

3.2 Global Naming in the DSS

To uniquely identify different DSS nodes, each node has a unique name within the distribution system. The name is constructed as a randomized 256 bit string, which guarantees uniqueness. Since the name is unique we avoid name collisions.

The DSS provide a global name schema for identifying data structures. The global name is constructed by combining the global identifier of the node
with a locally unique index.

4 Efficient Distribution of Immutable Data Structures

Immutable data structures are set to a value upon creation, and that value is never changed. Separating mutable and immutable data structures gives a local program a comprehensive structure. This separation can be used when the data structure is distributed for increasing efficiency. Immutable data structures exist in most programming languages, for example CONST in C, atoms and records in functional languages, or user defined data types, but can also be a class description or a procedure.

None of the systems in Section 2 presents an efficient and dynamic structure for handling immutable data structures. The common solution is to replicate immutable data, which provides the remote node with a local copy of the data that it can access without delay. The drawbacks of this strategy are:

**Memory usage** For each replica of a data structure that is sent to a remote node, a new copy is initiated. This can be potentially wasteful if the same replica is re-sent to the same node. Furthermore, this is not the behaviour if the operation is executed locally, e.g. if an entity instance is accessed locally, only a reference to this entity instance is passed. In the distributed case, a new instance of the entity will be created at each reference.

**Network usage** The data structure is replicated to all nodes that reference it, whether it is used or not. This can potentially result in excessive network usage since an entity instance can be transferred and installed at a node where it is never accessed. Another potential waste is when multiple instances of the same entity are transferred to the same node, as mentioned before.

There is not one strategy for distributing immutable data structures that will be the most efficient in all situations. The data structures can be categorized by size, frequency of usage and how sensitive the system is for waiting for the data structure. We call these categories distribution scenarios. There are different strategies that are optimal in the various distribution scenarios.

Real world examples could be:

**Atoms, Strings** These data structures are small so the overhead for replicating them among the nodes is negligible. We would rather not let our
Figure 3: The immutable data structures categorized

evaluation be delayed trying to save memory and network bandwidth on these data structures.

**Code on demand, Classes** Structural definitions of data structures, as class definitions, and code that is shared among nodes [16] introduce a group of shared entities that are moderate in size but need an identifier to not be duplicated.

**Files, Bitmaps** Large data structures that should be distributed with caution. The structure should allow partial access. The data structure is sparsely accessed and is considered too large to distribute. An extreme example could be a directory of files, where only the names and size need to be distributed. Distributing these data by replicating the complete files would produce an enormous overhead.

### 4.1 A Taxonomy of Immutable Data Structures

This taxonomy of immutable data structures is based on size, structure and usage pattern. The structure of the data is important if it is sparsely accessed, e.g. accessing on row in an array. The usage pattern of a data structure includes how sensitive it is to delay and if the data structure is accessed frequently. The data structures can be categorized into four different groups (Figure 3):

**Small Immutable Data Structures**

Small immutable data structures can be distributed freely because of their size. Since we want to avoid any delay when addressing these structures,
they should be distributed directly. Duplicated instances and re-sending of
the data structure is in most cases cheap, unless we experience some extreme
usage patterns, since the overhead for each message is large. Examples of
this type of immutable data structures are atoms and strings.

**Large Immediate Immutable Data Structures**

Large immutable data structures that are commonly accessed by many nodes
should be replicated eagerly, but duplicated copies of the same instance need
to be avoided. By naming these structures they can be identified and by keep-
ing a local record of present data structures, duplicates can be avoided. But
if the data structure is not present at the node, it is immediately requested
and installed locally. Before the data structure is fully installed there needs
to exist a coordination network between the originating and the remote node.
When the data structure is completely transferred the coordination network
can be dismounted. Examples of this type of immutable data structure are
code on demand or class descriptions.

**Large Delayed Immutable Data Structures**

Large delayed immutable data structures should not be distributed until they
are accessed. As above, by naming the structure, unnecessary duplicates and
transfers can be avoided. Since the data structure should not be transferred
to nodes until explicitly used, the node will just have a skeleton structure,
representing the actual entity. This skeleton will work as a proxy and when
it is accessed it requests the data from the originating node. While the
data structure is installed the calling thread\(^4\) has to be suspended. As be-
fore, there will be a coordination network until the complete data structure
is transferred. This type of data structure is for example files or bitmaps
that are infrequently accessed, but the ones that are accessed are done so
frequently.

**Very Large Immutable Data Structures**

For very large structured immutable data structures that are not frequently
accessed, it is more efficient not to replicate the data structure at all. The
data must have a structure that enables partial access or have operations that
return a result that is much smaller than the complete data structure. The
data structure is distributed as a proxy. When it is accessed the operation is
sent to the node where the data structure is located and executed there. The

---

\(^4\)Recall that we assume a PS with a notion of multiple threads.
result is then returned to the calling node. The coordination network will be present until the reference to the remote object is removed. The immutable data structure acts as a remote object. Examples of data structures are bitmaps and files that are structured and sparsely accessed.

4.2 Tools for Efficient Distribution of Immutable Data Structures

For realizing efficient distribution of immutable data structures we present a naming scheme and a proxy structure for the data structures. The naming scheme gives us a tool for avoiding duplication of replicas at a node. The proxy structure provides us with a tool for avoiding unnecessary replication.

Naming

By naming an instance of a data structure it can be replicated without creating an unnecessary duplicate. Each node has a local name table for keeping track of present instances. The name should be unique for each data structure instance, i.e., entity. When an entity is replicated to a node its name is looked up in the nodes name table to decide whether an entity instance is already present or if a new instance should be initialized.

A coordination network for each shared entity realizes distribution of mutable data structures in the DSS. The coordination network will keep track of all the replicas and a global garbage collector will remove replicas when they are no longer used. When there is only one copy left that copy is localized and any future replication will start by distributing this instance. Hence there can not exist multiple copies of the same instances of a data structure, neither local or in parallel on another node, if they are not included in the same coordination network.

Immutable data structures are localized when the structure is completed at the remote node. If the same data structure is sent to the same node at a later occasion, there is no possibility to identify the data structure resulting in that a new copy of the same instance is built. When concerned with small immutable data structures where all delay should be avoided, having additional copies is a plausible trade-off. But for larger immutable data structures this is unacceptable.

To avoid collisions in the name space, the name has to be unique within the distributed system. This is solved by using a global name in the DSS, see Section 3.2.
Naming on PS Level

In many PS that are supported by the DSS there already exists a possibility to name entities to avoid duplicate replication. But these names are only possible to use at a PS level and to efficiently avoid duplicated entities we need a name structure on the distribution level.

To avoid name collisions it is preferable to use the global names provided by the DSS even at the PS level.

Proxy Structure

To avoid that an immutable data structure is replicated to a node that does not use it, we introduce a proxy that represents the entity at a remote node. This proxy is small, i.e., cheap to distribute. Since we are distributing an immutable data structure, the only operation this proxy should support is a read operation. When an immutable data structure is shared, the proxy structure is marshaled and sent to the remote node. The remote node will then unmarshal the message and build the proxy. If the proxy is read the calling thread will be suspended until the complete data structure is locally present. The proxy can be equipped with an arbitrary distribution protocol, depending on what kind of data is distributed.

Using a lazy protocol that only sends the complete data structure when the proxy is accessed, will assure that the data structure is not replicated to a node that does not use it.

For larger immutable data structures a combination of naming the data structure and initially distributing it using a proxy structure can avoid unnecessary duplication in case of multiple replication.

4.3 Strategies for Distribution of Immutable Data Structures

It is important to choose the right distribution strategy when the data is replicated. These four strategies can be directly related to the different kind of data presented earlier:

Immediate Replication

The data structure is replicated to the remote node immediately when the reference is created. The data structure is completely built at the remote node and the distribution structure is dismounted right away.

Using this strategy on a named entity will not reduce network traffic, since the complete entity is always sent together with the name, even if it is
already present at the receiving node. The benefit is still that duplication is avoided.

This strategy is ideally used for small immutable data structures, since the extra network traffic and memory usage is weighted against letting the thread wait.

**Eager Replication**

A proxy representative of the data structure is initially transferred to and built at the remote node. The remote node will receive the complete data structure when the proxy is built. When the complete data structure is installed the distribution structure is dismounted.

Using this strategy on a named entity is ideal. The size of the name and the proxy structure that is initially sent is small. If a structure is remotely present, the cost of re-sending that entity is minor.

This strategy is preferable to use on large immutable data structures where the probability that the receiving node will actually use the data structure is high.

**Lazy Replication**

As for the Eager replication model, a proxy representative is initially built at the remote node. But the data structure will not be completed until it is actually used.

This strategy and a named entity can be profitable combination to avoid unnecessary replication. Since the data is not transferred unless it is used and naming guarantees that data is not duplicated, both network traffic and memory usage is reduced.

This strategy is a good choice distributing large data structures where replication should be avoided.

**Remote Procedure Call (RPC)**

A proxy is distributed to represent the data structure at remote nodes. All operations on the data structure are forwarded to the node where the entity is located. The strategy requires that the entity has a structure that allows partial access, e.g., arrays, or that the result of the operation is smaller than the complete data structure.

This strategy will keep a coordination network alive as long as it is distributed so there is no need for naming the entity.

Data structures that are very large and not frequently used would benefit from this strategy. It is also a good choice for keeping down network traffic
and memory usage, but it produces more coordination traffic than the other strategies.

5 Implementation

The DSS is modularized with well-defined interfaces separating different aspects of the distribution support. The API is separated from the protocol layer and the protocol layer is separated from the message layer, as described in Section 3. Within each layer it is very easy to add functionality, i.e., new protocols. We have added new protocols without altering the messaging layer. Adding and altering data structures is also easy and the choice of coordination protocol can be changed at any point.

5.1 Protocols

The choice of protocol is dependent on what kind of data is to be shared. Large data structures should preferably be named and combined with the lazy or the eager protocol for avoiding unnecessary network traffic. The name is matched against the local name table and the data structure is only transferred if it is not already present. The eager protocol is preferably used if the thread is sensitive to delays. On the other hand, the lazy protocol only transfers the data structure if it is being used. The immediate protocol is ideally used for small unnamed data structures. Naming data structures will not reduce network traffic, since the data is always sent, but will save memory space since duplicated instances will be removed.

RPC is implemented using the Simple Channel protocol provided by the DSS. This is efficient to use for very large structured data structures that are not frequently used.

The protocols are described using a sequence diagram of the interaction between the parts of the system. An instance of a data structure, i.e., an entity, is instantiated at a process. This entity instance is then replicated or remotely invoked. All operations are done through the AE when the entity instance is distributed. A skeleton structure is built at the remote node when the data is replicated. This skeleton structure is then either completed or it just transfers operations to the original entity instance. To make the skeleton complete the entity instance is transferred to the remote node and installed there as a local entity instance. In the sequence diagram there is a thread at the remote node that invokes the object with an operation.
Lazy Protocol

The protocol is displayed in Figure 4. Initially (1) the package containing information about the entity is sent to the remote process A. When the package arrives at process A (2) it is unmarshaled. With the information that is received an AE is built. This AE represents the entity and contains synchronization information described in Section 3. At this point process A only contains structural information about the entity, and it has to be completed before it can be accessed. The AE is passive until the thread makes an entity operation (3).

The entity operation from the thread in the PS is translated to an abstract operation (4). The entity is immutable so the only abstract operation available is a read operation. The operation is then passed to the AE. Since the entity has no data present, the AE sends a request using the coordination network. The coordination manager forwards the request to process B (5). While the AE waits for the entity data to be transferred and installed, the AE instructs the PS to suspend the thread (6). The AE at process B receives the request and it retrieves (7) the data from the entity instance at process B. The data is marshaled and the package is sent to process A (8). When the package is received it is unmarshaled and the AE installs the contents making the entity instance complete (9). The AE then tells the PS system to resume the thread (10). Since the immutable entity is completely present at process A it is localized, i.e., the entity is made local by dismounting the AE and all operations are from now on made directly on the local entity.

Figure 4: A sequence diagram of the lazy protocol
The thread is resumed and the operation is carried out (11).

![Sequence diagram of the eager protocol](image)

**Eager Protocol**

The eager protocol is similar to the lazy protocol as it first transfers and installs an AE and later completes the entity. The difference is that the entity will automatically be completed when the AE is built, i.e., no activity is required from the PS.

The eager protocol, displayed in Figure 5, works as follows; first, a package containing information about the entity is transferred from the node where it is currently present, i.e., process B in the Figure (1), to process A. The information includes what kind of AE to build, what kind of synchronization protocols \(^5\) to use, and the identifier of the originating node. When the package arrive at process A (2) it is unmarshaled. With the information that is received an AE is built. This AE represents the entity and contains synchronization protocols described in Section 3. This far the process A only contains structural information about the entity that has to be made complete before it can be accessed.

Directly after the AE is built it sends a request to process B (3), through the coordination network, for the data to complete the entity. The AE at process B receives the request and it retrieves (5) the data from the entity instance at process B. The data is marshaled and the package is sent to

---

\(^5\) Protocols for coordination, memory management and consistency.
process A (7). When the package is received it is unmarshaled and the AE installs the contents for completing the entity (9). Since the immutable entity now is completely present at process A it is localized, i.e., the entity is made local by dismounting the AE and operations can now be made locally.

If a thread in the PS should make an entity operation (4) while the AE is busy fetching the entity data, that operation will be handled the same way as for the Lazy protocol.

![Sequence diagram of the immediate protocol](image)

**Immediate Protocol**

The immediate protocol will make the remote entity instance complete directly when it is created. The protocol is displayed in Figure 6.

When the AE at process B is marshaled, it also retrieves (1) the entity data. This data is marshaled together with the AE information. The package is then forwarded to process A (2). As for the lazy and the eager protocol, the AE is built (3) using the received information. Directly after the AE is built the entity is completed (4) with the data that was piggybacking the received package. The AE is dismounted when the entity is completed. The entity operations (5) are then done completely locally.

Creating the AE communication structure might seem a bit wasteful, but the overhead is small and it lets us use the same AE model in all the protocols.

**Simple Channel Protocol**

The simple channel will work as a tunnel that forwards all operation to the node where the data structure is present and then returns the results to the
calling node.

The AE information is transferred (1) to process A where it is built (2). When the thread executes an operation on the entity (3), this operation is translated to an abstract operation (4). The abstract operation is then forwarded (5) to process B, where the entity is present. The calling thread is suspended (6) while awaiting the results. The operation is executed (7) on the data structure and the result is sent to process A (8). The result is then returned to the calling thread (9), which is resumed (10). The coordination network will be kept alive until the AE at process A is removed.

5.2 Named Data

Instances of distributed entities can be named with a globally unique name. By naming instances we can avoid having multiple replicas of the same instance at a node. As can be seen in Figure 8, the name is connected to the entity and not the AE, so the name remains when the entity is localized.

When a new instance of a named data structure is created, it is provided with a unique name from the distributing system. The name and a reference to the named entity is stored in a local name table at each node.

The first issue when distributing a named entity is to transfer the name, making it possible for the receiving node to check whether the entity is present or not. This is independent of which protocol that is used, the name is always transferred first. When an entity is distributed, a message containing information about the entity is sent to the remote node. The name is marshaled
and put in front in this message. After that, information about the AE, protocol specific information, etc., is marshaled and attached to the message. This is in case the entity is not present and a new structure has to be built.

Figure 9 shows the scenario where a named entity is received at a node where it is not currently present. The data is received (1) by the DSS. The DSS then tries to locate the named entity by looking up the name (2) in the local name table. The name is not resolved and this response (3) is sent to the DSS. Since the entity is not present the DSS initiates the creation of an AE (4) representing the entity. When the AE is created (5), the name and a reference to the entity is added to the name table (6), and the entity is returned.

In Figure 10 we display the scenario when the entity is available. As before, the DSS receives a message with entity information (1). The DSS performs a lookup on the name in the name table (2). Since an instance of
the entity is already present at the node, the response (3) from the name table contains a reference to that instance. The reference to the existing instance is returned and the message buffer is flushed (4).

Using this structure with unique names and local name tables ensures that an instance of a named data structure is not duplicated at a process.

5.3 Package Format

Distributing data structures requires an efficient package format. Our requirements when it comes to distributing immutable data structures are that it is dynamic and streamlined. We need a package format that allows us to add and remove information depending on what kind of distribution protocol that we use.

All protocols have some information in common that should always be included when an entity is initially distributed. The structure is displayed in Figure 11. This base package (1) contains information about origin, which protocol that is used for synchronization, etc.

For the named data structures it is crucial that the name is enclosed in an initial package. Since the name is the first thing that is checked when a package arrives in order decide whether an entity is present or not, the name (2) has to be placed in front of the base package (1). The name is of fixed size so the receiving node has no problem identifying it.

The base package can also be extended to satisfy certain protocols. This is done by letting the protocol encapsulate arbitrary data at the end of the

Figure 11: The package structure
The receiving node will build the AE (described in Section 3) and then let the AE handle the piggybacked data. For example, the immediate protocol uses this package to piggyback the data in the entity. When the AE is built, the immediate protocol uses the data that is received for completing the entity.

![Diagram of package structure]

**Figure 12: The package structure**

**Packaging**

Transferring an entity to a remote node starts by asking the entity to marshal itself. The DSS initiates the AE to be marshaled. The marshaling procedure conducted by the AE is displayed in Figure 12. In the Figure, the dash-dotted line represents the buffer that the package is sent on. The AE initially (1) requests the entity to marshal its name to the buffer. Next, the DSS marshals the AE information (2) to the buffer. The protocol (3) may then add arbitrary data to the buffer, for instance additional entity information.

We present the following pseudo code to show how the packing is done in the DSS. The code is written in a high level language. The pseudo code displays the marshaling and unmarshaling of the package.

```java
void marshal(Buffer buf) {
    if (entity.isNamed()) then
        buf.marshal(entity.isNamed());
```

---

6Entities are transferred as a result of operations on other entities, i.e., an immutable could be an argument to an RPC.

7This is obviously just the case for named data structures.
buf.marshall(entity.name)
fi
buf.marshall(abstractEntity)
protocol.marshallProtocolInfo(buf)
}

void unmarshall(Buffer buf) {
  Bool isNamed = buf.unmarshallBool()
  if isNamed then
    GlobalName name = b.unmarshallName()
    Reference localRef = dss.nameTable.lookup(name)
    if localRef != NULL then
      AbstractEntity ae = buf.unmarshallProxy()
      entity = new Entity()
      unmarshallProtocolInfo(buf)
    else
      entity = localRef
      buf.flush()
    fi
  else
    AbstractEntity ae = buf.unmarshallProxy()
    entity = new Entity()
    unmarshallProtocolInfo(buf)
  fi
}

The marshaling process straightforward, the different parts are marshaled to the buffer, as displayed in Figure 12. The unmarshaling process is somewhat more complicated. We first have to check if the entry is named, then if the name exists, i.e., an instance of the entity is already present.

The package format is flexible and allow the protocols to append information to the distribution package. This can for instance be used for the immediate protocol to append entity data.

6 The StringVec Data Structure

To clarify how the immutable data structures can be constructed and how it interacts in the DSS we here present our StringVec data structure. The StringVec is an implementation of a vector that contains arbitrary strings.

The StringVec is a completely local structure and it only supports the operation getString(pos) that returns the string at position pos in the
Figure 13: The distribution support for the StringVec data structure

vector. This data structure can be considered as an arbitrary immutable data structure that is distributed and thereby needs to be extended with distribution support.

In Figure 13 we present a UML [17] diagram of distribution structure for the StringVec data structure. The diagram is simplified to emphasize how the data structure interacts on the different levels. This distribution structure can be used for arbitrary data and it can easily be customized for any immutable data structure. Changing coordination protocol is done by a setting an attribute when creating a new distributed instance of the StringVec. To add distribution support for the StringVec data structure the following data structures are added:

MapStringVec This is used as an interface for the user for interaction with the StringVec. The MapStringVec keeps track of whether the StringVec is locally present or not. The function getString() will be performed directly on the StringVec if it is present, otherwise the MapStringVec will forward the call to the StringVecMediator. The MapStringVec can also contain a name.

StringVecMediator This structure is an interface between the

*As can be seen in the evaluation code in Appendix A.
MapStringVec and the ImmutableAbstractEntity. The StringVecMediator translates a data structure specific operation to an abstract operation that is passed to the ImmutableAbstractEntity. The StringVecMediator also responds to remotely initialized calls that are forwarded by the ImmutableAbstractEntity. The CallbackRead() is called when the entity is accessed by an RPC. This function will perform a getString() on the StringVec and then return the response to the calling node.

The RetrieveRepresentation() operation marshals the StringVec and forwards it to the ImmutableAbstractEntity that transfers it to a remote node. The corresponding operation at the receiving node is InstallRepresentation(), that installs the StringVec.

ImmutableAbstractEntity Together with the StringVecMediator, the ImmutableAbstractEntity form what we in Section 3 presented as an AE. The ImmutableAbstractEntity is a general construction that is the same for all immutable data structures. It is in the ImmutableAbstractEntity the Coordination protocol is set. The ImmutableAbstractEntity forwards abstract operations, entity retrieval, and installation operations to the StringVecMediator.

The RetrieveEntityState() function is used for retrieving the state of an entity when it is replicated to a remote node, but since the data is stateless the function only requests the entity representation from the StringVecMediator. At the remote node the corresponding function is InstallEntityState() that just forwards entity data to the StringVecMediator for installation.

6.1 Distribution

Operations on a distributed immutable data structure are done in one of two ways, either the data structure is replicated to the calling node or an RPC is performed and the result is returned.

RPC

The RPC is initialized when the getString() in the MapStringVec is called at a node where the StringVec is not present. In the MapStringVec the getString() function checks if the data is present, and since this is not the case, it calls the StringVecMediator (1) to fetch the data remotely. The StringVecMediator will translate the data structure specific getString() to a general abstract read operation in the ImmutableAbstractEntity (2).
Figure 14: RPC on a StringVec

The ImmutableAbstractEntity decides how the operation should be carried out according to the coordination protocol. In this case we perform an RPC so it transfers the abstract read operation to the node where the entity is present and applies the abstract operation (3). Data is retrieved from the entity by the operation CallbackRead() on the StringVecMediator (4). This function calls the getString() in the MapStringVec (5), that executes the getString() operation on the StringVec (6). The result is then marshaled and sent back over the connection to the calling node, where the answer is returned and the thread is woken up.

Replication

The Immediate, Eager, and Lazy protocol all replicate the data to the remote node. We will only display the Lazy protocol since the interaction is very similar to the other protocols, but the Immutable and the Eager protocol are able to skip a few steps since the replication is automatic.

The Lazy protocol will only replicate the data if the remote node tries to access it. This is displayed in Figure 15. The MapStringVec will first check if the StringVec is present, otherwise it calls the getString() in the StringVecMediator to retrieve the data (1). This will suspend the calling thread. Since the coordination protocol is set in the ImmutableAbstractEntity, this procedure is identical to the RPC case. The StringVecMediator translates the getString() to an abstract opera-
Figure 15: Replicating a StringVec using the Lazy protocol

tion (2). Because of the Lazy protocol the ImmutableAbstractEntity sends a request for a replica of the entity (3). The ImmutableAbstractEntity receives a request for retrieving the entity state, but since the entity is stateless the request is forwarded to the StringVecMediator as an operation for retrieving the representation of the StringVec (4). The complete StringVec is now marshaled (5) and transferred to the calling node’s ImmutableAbstractEntity, which receives the stateless entity. Since the ImmutableAbstractEntity contains no information about the StringVec structure it passes the received data to the StringVecMediator (6). The StringVecMediator installs the StringVec (7). Then the getString() operation is performed and the thread calling the MapStringVec is woken up (8).

The Immediate protocol will retrieve the entity (4) at the same time the AE is marshaled and then installs it in the same way as described above (5-8).

In the Eager protocol it is the protocol on the remote node that initializes the entity retrieval (3). The rest of the procedure is carried out the same way as for the Lazy protocol (4-7).
7 Evaluation

From Section 4 it should be clear that there is no single strategy to efficiently distribute immutable data structures. To make the distribution efficient there is a need for different distribution strategies for different immutable data structures. Note that we by different include structure, size, and usage pattern. The evaluation is conducted by using four different distribution strategies in four different scenarios, concluding that each scenario requires an adapted distribution strategy.

The evaluation is done by transferring a number of StringVecs with different size in the different scenarios. A distributed future is sent along with the immutable data, so the host\(^9\) will know when the client\(^10\) is done. The code used in the evaluation is in Appendix A.

The evaluation was done using two computers at SICS. The client node run on an Intel P4 2.0 GHz with 512 MB RAM using RedHat 9.0, and was located in Kista, Sweden. The host node run on an Intel P2 350 MHz with 512 MB RAM, using RedHat 9.0, and was located in Uppsala, Sweden. The computers were connected with a WAN connection with an average ping time of 3.8 milliseconds. The version of the DSS was a CVS snapshot dated 2004-03-08.

We present four different distribution scenarios for immutable data structures that resemble real life situations. The scenarios have different requirements and the size of the data structures differs.

**Scenario 1** In the first scenario we have many small data structures that should be distributed so that the delay at the remote node is minimized.

**Scenario 2** In this scenario we distribute larger data structures that are all used for computations at the remote node. The objective is to minimize the delay at the remote node but without wasting memory space and network traffic, i.e., only one copy of each instance should be present.

**Scenario 3** Here we also share larger data structures, but not all of these are used for computations at the remote node. The objective here is to minimize memory usage and network traffic. Delay is of lesser concern.

**Scenario 4** In this last scenario we share an even larger data structure, but it is very sparsely used. The objective is to keep down memory usage and network traffic. It is unnecessary to transfer the complete data structure since only a small part of it is actually used.

---

\(^9\)The host in this system is where the all the data is initially.

\(^10\)The client in this system is the receiver and user of the data.
7.1 Evaluation setup

A test program is constructed for each scenario to emphasize its characteristics. The test program has a host and a client part. The host creates the shared entities and initiates the distribution. We have used the StringVec data structure presented in Section 6 in our test. The functionality differs in the different scenarios:

Scenario 1

**Host side:** The test data is constructed by 100 StringVec entities of size 20x5\(^{11}\) (100 bytes). This makes our test data 10 000 bytes.

**Client side:** The client program will use all of the 100 entities and read all the 20 rows.

Scenario 2

**Host side:** The test data is constructed by 30 StringVec entities of size 20x100\(^{12}\) (2 000 bytes). This makes our test data 60 000 bytes, but all entities are duplicated, so the amount of unique data is 30 000 bytes.

**Client Side:** The client program will use half of the 30 entities and read half of their 20 rows of data.

Scenario 3

**Host side:** The test data is constructed by 30 StringVec entities of size 20x100\(^{13}\) (2 000 bytes). This makes our test data 60 000 bytes, but all entities are duplicated, so the amount of unique data is 30 000 bytes.

**Client Side:** The client program will only use 3 of the 30 entities and read half of the rows of data.

Scenario 4

**Host side:** The test data is constructed by one StringVec entity of size 4000x10\(^{14}\) (40 000 bytes). This makes our test data 40 000 bytes.

---

\(^{11}\)20 rows with 5 bytes strings.  
\(^{12}\)20 rows with 100 bytes strings.  
\(^{13}\)20 rows with 100 bytes strings.  
\(^{14}\)4000 rows with 10 bytes strings.
<table>
<thead>
<tr>
<th>Distribution protocol</th>
<th>Host to Client</th>
<th>Client to Host</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>bytes</td>
<td>messages</td>
</tr>
<tr>
<td>Immediate</td>
<td>35 452</td>
<td>4</td>
</tr>
<tr>
<td>Eager</td>
<td>60 993</td>
<td>203</td>
</tr>
<tr>
<td>Lazy</td>
<td>60 993</td>
<td>203</td>
</tr>
<tr>
<td>RPC</td>
<td>918 452</td>
<td>4004</td>
</tr>
</tbody>
</table>

Table 1: In Scenario 1 we transfer 100 unique StringVecs of the size 20x5 from a Host to a Client. In all the 100 StringVecs all the 20 rows are read.

<table>
<thead>
<tr>
<th>Distribution protocol</th>
<th>Host to Client</th>
<th>Client to Host</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>bytes</td>
<td>messages</td>
</tr>
<tr>
<td>Immediate</td>
<td>62 798</td>
<td>4</td>
</tr>
<tr>
<td>Eager</td>
<td>33 818</td>
<td>19</td>
</tr>
<tr>
<td>Lazy</td>
<td>33 818</td>
<td>19</td>
</tr>
<tr>
<td>RPC</td>
<td>49 317</td>
<td>153</td>
</tr>
</tbody>
</table>

Table 2: In Scenario 2 we transfer 30 StringVecs that are duplicated and of the size 20x100. In half StringVecs 10 of the rows are read.

**Client Side:** The client program will only read one row in the entity.

Measuring the time it takes to distribute the data structures is very dependent on the hardware. Since our test scenarios are built to emphasize distribution characteristics rather than existing programs, the actual time it takes to distribute is uninteresting. It is more interesting to measure the number of messages and the amount of data that is passed between the nodes. If the distributed entities are small compared to the network bandwidth, the size of each message is less important than the amount of messages.

### 7.2 Results

The results presented here were obtained when distributing test data as presented in Section 7.1. To coordinate the test program a distributed future is used, which introduces a small overhead in the communication but this overhead is the same in all the tests.

In Scenario 1 the Immediate strategy is the far most effective, as can be seen in Table 1. The Immediate strategy produces very little overhead in network traffic when it replicates the data. Since all entities are unique we have no benefit of naming the entities, but the overhead for this is merely about 1000 bytes. The Lazy and the Eager strategies produce the same amount of network traffic, but the Lazy strategy has to suspend the client
for each entity, while the data is fetched. All of the data in each entity is read and this results in an enormous overhead for the RPC strategy\textsuperscript{15}.

In Table 2 our different distribution strategies are evaluated in Scenario 2. Comparing just the Immediate and the Eager strategy, we can observe that the Eager transfers only half the amount of data, because it avoids re-sending duplicates, but it will on the other hand use more messages. In this scenario, all data structures are read so the Lazy and the Eager strategy will produce the same amount of messages, but the Lazy strategy will always suspend the client when data is fetched, resulting in better performance for the Eager strategy. In this scenario, the RPC strategy is clearly a bad choice since it produces an overhead in both bytes and messages.

The results in Scenario 3, displayed in Table 3, show that the Lazy strategy is most beneficial. Both the Immediate and the Eager strategy transfer all the entities to the client but only one tenth of them are used. Half of the data in the entities that are transferred is read and this results in an overhead for the RPC strategy.

Using RPC is the best choice in Scenario 4 as can be seen in Table 4. When there is a big data structure that we just want to access a small part

\textsuperscript{15}The number of messages sent is correct, but the DSS seems to produce a lot of overhead on the RPC-calls when each call is over 200 bytes and the data is only 5 bytes.
of, the RPC is superior. All the other strategies have about the same result since the complete entity has to be transferred before it can be read.

The scenarios we have presented and evaluated in this section is distribution scenarios that are common for immutable data structures in a distributed environment. Our different distribution strategies perform well in some scenarios and worse in others. This leaves us with the conclusion that there is no single strategy to efficiently distribute immutable data structures.

8 Conclusion

We have conducted a small survey on DSM systems and how they handle immutable data structures. These systems present advanced techniques for distributing mutable data structures, but immutable data structures are disregarded. To efficiently distribute these structures their characteristics should be taken into consideration.

Immutable data structures include a wide range of stateless data with different size and usage. There is no single solution to efficiently distribute all of these which calls for a dynamic structure.

We identify four different distribution scenarios for immutable data structures and present a dynamic solution for handling them. The data structures can be categorized on their size and how they are accessed. By this categorization we identify four kinds of immutable data structures that all need different distribution strategies to be handled efficiently.

Our solution is to used one of the following distribution strategies; Immediate, Eager, Lazy or RPC. The Eager and Lazy strategy is the Eager/Lazy protocol combined with naming the entities. These strategies have been implemented as part of the DSS.

The burden for the user is reduced by providing an abstract interface that hides protocol specifics. Adding customized protocols and distributed data structures is easy due to the modular structure of the DSS. The API provides the user with an easy to use interface with a great variety of protocols to choose from. This is done without exposing the user to the complexity of data consistency, memory management, etc. Distribution of immutable data structures can be done simple and efficiently.

9 Acknowledgments

A very big thank you to my supervisor Erik Klintskog for all help and support. Without his prior work on, and insights in, the DSS this thesis would not
have been possible.

References


A The evaluation program

Here we present the code that we used when we evaluated the performance.

A.1 The Host node

/* immutable host
 *
 * Authors:
 * Per Sahlin (sahlin@sics.se)
 *
 * Contributors:
 * optional, Contributor's name (Contributor's email address)
 *
 * Copyright:
 * Per Sahlin, 2004
 *
 * Last change:
 * $Date: 2004/03/15 13:05:08 $ by $Author: sahlin $
 * $Revision: 1.1 $
 *
 * This file is part of Mozart, an implementation
 * of Oz 3:
 * http://www.mozart-oz.org
 *
 * See the file "LICENSE" or
 * http://www.mozart-oz.org/LICENSE.html
 * for information on usage and redistribution
 * of this file, and for a DISCLAIMER OF ALL
 * WARRANTIES.
 *
 */

#include <stdio.h>
#include <string.h>
#include <fcntl.h>
#include <stdlib.h>

// For inet_addr
#include <sys/socket.h>
#include <netinet/in.h>
#include <arpa/inet.h>

#include "dss_object.hh"
#include "pthread_map.hh"

#include "map_datatypes.hh"
#include "map_ticket_mechanism.hh"
#include <sched.h>
#define RESULT_FILE "res_host1.txt"

void* program(void *arg) {

    registerThread();

    MapStream *s = new MapStream();
    MapPort *mp = new MapPort(s);
    // Initial communication with the Host
    char ** argv = reinterpret_cast<char**>(arg);
    char *str = entityToTicket(mp);
    printf("Process:%d\n",getpid());
    printf("Ticket:\n");
    printf("%s\n",str);
    FILE* tick = fopen(reinterpret_cast<char*>(argv[1]),"w");
    fprintf(tick,str);
    fclose(tick);
    delete str;

    // Setting up the scenario
    int readfreq;
    int readElements;
    int scenario = atoi(reinterpret_cast<char*>(argv[2]));
    switch (scenario) {
    case 1:
        readfreq=1; // read all StringVecs
        readElements=1; // read 20 rows in the StringVec
        break;
    case 2:
        readfreq=2; // read half the StringVecs
        readElements=2; // read 10 rows in the StringVec
        break;
    case 3:
        readfreq=10; // read 1/10 of the StringVecs
        readElements=2; // read 10 rows in the StringVec
        break;
    case 4:
        readfreq=10; // read 1/10 of the StringVecs
        readElements=20; // read 1 row in the StringVec
        break;
    default:
        Assert(0);
    }

    int j=0;
    int i= 0;
    // Receive the StringVecs
    MapTuple* data = static_cast<MapTuple*>(s->getValue());
int ar = data->getArity();
for(i=0; i<ar-1;i++) {
    // Each StringVec
    MapStrVec* ssv = static_cast<MapStrVec*>(data->getElement(i));
    if((i % readfreq==0)) {
        for(j=0; j<20 ;j++)
            if (j % readElements==0)
                ssv->getString(j); // Read row
    }
}
MapVariable* reply = static_cast<MapVariable*>(data->getElement(ar-1));
// Save results to file
FILE* res = fopen(RESPONSE_FILE,"a");
// nobs = number of bytes sent
// noms = number of messages sent
fprintf(res,"%d ; %d ; %d \n",
    scenario, g_mcu->nobs(), g_mcu->noms( )
);
fclose(res);
// Bind variable and the host will continue
reply->opBind(c2m_int(scenario));

return 0;
}

int main(int args, char* argv[]){
    if(args != 4)
        {
            printf("immutable_server.out <FILE> <SCENARIO> <T0_IP>\n”);
            _exit(0);
        }
    pthread_t prog;
    int ip = static_cast<int>(ntohl(inet_addr(argv[3])));

    // Initializing the DSS
    bool s = false;
    startProcessMCU(new AppMediationObject(), ip ,s);
    pthread_create(&prog, NULL, &program, argv);
    pthread_join(prog,NULL);
    return 0;
}
A.2 The Client node

/* immutable client
 *
 * Authors:
 * Per Sahlin (sahlin@sics.se)
 *
 * Contributors:
 * optional, Contributor’s name (Contributor’s email address)
 *
 * Copyright:
 * Per Sahlin, 2004
 *
 * Last change:
 * $Date: 2004/03/15 13:05:09 $ by $Author: sahlin $
 * $Revision: 1.1 $
 *
 * This file is part of Mozart, an implementation
 * of Oz 3:
 * http://www.mozart-oz.org
 *
 * See the file "LICENSE" or
 * http://www.mozart-oz.org/LICENSE.html
 * for information on usage and redistribution
 * of this file, and for a DISCLAIMER OF ALL
 * WARRANTIES.
 *
 */

#include <stdio.h>
#include <string.h>
#include <fcntl.h>
#include <unistd.h>
#include <stdlib.h>

#include "dss_object.hh"
#include "pthread_map.hh"
#include "map_datatypes.hh"
#include "map_ticket_mechanism.hh"
#include <sched.h>
#include <sys/time.h>

#define RESULT_FILE "res_scen1.txt"
void* program(void* arg){
    char *ticket = new char[300];
    char **argv = reinterpret_cast<char**>(arg);

    registerThread();

    // Setting up communications
    FILE* tick = fopen(reinterpret_cast<char*>(argv[1]),"r");
    fscanf(tick,"%s",ticket);
    printf("Connecting to: %s\n",ticket);
    MapPort *cp = static_cast<MapPort*>(ticketToEntity(ticket));

    // Choosing protocol and whether the entity is named.
    bool name;
    ProtocolName pn;
    int scenario = atoi(reinterpret_cast<char*>(argv[2]));
    switch(scenario) {
    case 1:
        name = false;
        pn = PN_IMMEDIATE;
        break;
    case 2:
        name = true;
        pn = PN_IMMUTABLE_EAGER;
        break;
    case 3:
        name = true;
        pn = PN_IMMUTABLE.LAZY;
        break;
    case 4:
        name = false;
        pn = PN_SIMPLE_CHANNEL;
        break;
    default:
        Assert(0);
    }

    // Setting up the size of the StringVec
    int x = atoi(reinterpret_cast<char*>(argv[3]));
    int y = atoi(reinterpret_cast<char*>(argv[4]));
    int nr_strv = atoi(reinterpret_cast<char*>(argv[5]));

    int i=0;
    // Creating an array of StringVec's
    MapBaseType **vec = new MapBaseType*[2*nr_strv+1];
    for (i=0;i<2*nr_strv;i++) {
        // Creates a StringVec with a choice if it should be named and protocol.
        MapStrVec *ssv = new MapStrVec(new StringVec(x,y), name, pn);
        ...
// The StringVec is duplicated (Scenario 2-3)
vec[i] = ssv;
vec[i+nr_strv] = ssv;
}

// A distributed future is created and sent.
MapVariable * reply = new MapVariable();
vec[(2*nr_strv)] = reply;

MapTuple *data = new MapTuple("", vec, 2*nr_strv+1);

cp->send(data); // The data is transferred.
reply->wait(); // Wait for response
int r_scen = m2c_int(reply->getVal());

// Save the result to file
FILE* res = fopen(RESULT_FILE,"a");
// noms = Number of messages sent
// nosb = Number of bytes sent
fprintf(res,"%dx%d nr:%d ; %d ; %d ; %d ; %d\n", x,y,nr_strv, r_scen,scenario, g_mcu->nobs(),g_mcu->noms(), msec);
fclose(res);

return 0;
}

int main(int args, char* argv[])
{
    if(args != 6)
    {
        printf("immutable_access.out <FILE> <SCENARIO> <X> <Y> <NR> <T0_IP>\n");
        _exit(0);
    }

    pthread_t prog;
    int ip = static_cast<int>(inet_addr(argv[6]));

    // Initializing the DSS
    bool s = false;
    startProcessMCU(new AppMediationObject(), ip ,s);
    pthread_create(&prog, NULL, &program, argv);

    pthread_join(prog, NULL);
    return 0;
}