A Library-Based Distributed Shared Memory System Implemented Using IP Multicast

Andrew J. Weber

September 8, 1999

Abstract

Distributed computing systems made up of a collection of computers require effective communication between the participating computers. The two fundamental models of communication are the message passing model and the shared memory model. The shared memory model can be further broken down into hardware-based, page-based, and library-based implementations. NCS (Network Chunk System) implements a library-based shared memory model of communication using IP multicast as the underlying transport. NCS provides for several dynamically configurable data replication behaviors. Each named data element, or chunk, may be configured to be replicated in any one of the supported ways at any given time. This provides flexibility in configuring NCS for performance and fault tolerance. This paper presents the overall design of NCS and introductory guide to using NCS.

1 Introduction

Communication in Beowulf and NOW (Network of Workstations) clusters currently have a variety of communication methods available. For the message passing model of communication there is PVM and MPI/LAM as well as TCP sockets. Beowulf clusters can also choose among page-based shared memory implementations such as DIPC [Kar99] and JIAJIA [HST99]. Also available are some library-based implementations using point to point transports such as pvmsync [Pit99]. NCS adds to the available choices for Beowulf and NOW clusters a library-based distributed shared memory system that is implemented using a multicast transport.

NCS is designed to replace a previous software distributed shared memory implementation (the Coral module DSS) [Inf98] that used point to point message passing (TCP) as its underlying transport.

1.1 Design Goal (Requirements)

The general purpose of NCS is to provide a data object based distributed shared memory system. The following design issues and features are considered.

Simplicity

NCS must be as easy to use as possible while still providing the necessary flexibility to be of practical use. This is a general requirement of all software but it is important so it is explicitly stated here.

Portability

NCS must be portable to many platforms, including platforms that are not POSIX compliant [IEE90]. In particular NCS must be portable to a variety of UNIX/UNIX-like platforms such as Linux, SGI's IRIX, Sun's Solaris, and HP's HP-UX as well as Microsoft's Win32 platforms such as Windows NT.

Scalability

NCS must scale well in a cluster environment such as a Beowulf [RBMS97] site. This means that NCS must be usable in clusters ranging from several nodes in size to several hundred nodes in size.

DSS Compatibility

Due to an installed base of DSS users as well as serving as a design point of functionality, NCS must provide a drop in compatibility class for the class DSS_Connection [Inf98].

Fault Tolerance

NCS must provide the ability to survive the fail-stop [SS93] loss of participating nodes. NCS must have a defined behavior in the case of a network partition. NCS must provide the configurable capability to not lose data in the event of fail-stop node loss. This configurability must provide the application developer the ability to make tradeoff decisions between data survivability and availability with service overhead.

Architectural Flexibility

NCS provides the ability to architect an application in any of many different ways such as traditional client/server, traditional peer to peer, DSS style [Inf98] server clusters, proxy configurations, or hybrids.

1.2 Design Decisions and Tradeoffs

Library Implementation

Implementing NCS as a library was done in order to meet several of the requirements. The primary reason was for portability since a virtual memory manager implementation, also known as a page-based implementation [CDK94] inherently requires system dependent code and usually kernel modifications such as is the case with Ivy [LH89] and Munin [CBZ91].

Another benefit of a library implementation is the ability to reduce and often eliminate false sharing. False sharing is when data is unnecessarily transferred between machines. This is often associated with the transfer of a whole page of memory in a virtual memory implementation when only a fragment of the page is used [CDK94]. Since a library implementation can provide user defined variable sized pages, called chunks in NCS, the user has the ability to share only what is necessary.

Consistency Model

From a direct memory access point of view NCS provides a common form of relaxed consistency called release consistency [CBZ95] [GGH91]. Release consistency requires the explicit use some synchronization mechanism, and in the case of NCS this is the open and close operations. Therefore, release consistency is provided by making the memory consistent when a close operation is done.

However, from the point of chunk accesses, especially when a chunk access is considered atomic as is the case with the NCS_Simple interface, the consistency model is only slightly weaker than sequential consistency [Lam79]. Access at this level is a form of local sequential consistency called cache consistency [CBZ95].

Single Writer or Multiple Writer Sharing

The current version of NCS has only a single-writer-multiple-reader access implementation. However, a multiple-writer-multiple-reader algorithm can readily be added as another option by plugging in the algorithm to the request queue processing code, and the necessary merge routine (depending on the algorithm chosen) to the chunk update code. This is one area of future work.

IP Multicast as Underlying Transport

Using a true multicast or broadcast based transport is much more scalable than using a point to point based transport. This is especially true in the case of sending or replicating the same data to multiple destinations. In the case of a multicast transport this data only needs to hit the wire once while in the point to point transport case the data will hit the wire once for every destination. However, a tradeoff is that a multicast implementation can have somewhat more CPU overhead due to the fact that every participating machine will receive all packets [CDK94]. However, in practice this overhead is very small compared to the basic communication costs of current networks. Also, the latency in low traffic scenarios may be higher in a multicast implementation due to the message stability requirement of the transport required by the sequential consistency implementation [Jal94].

State Machine Approach

The fault tolerance requirement that data must be able to survive the loss of participating nodes means that NCS must support data replication. There are three main approaches to designing a data replication system; voting, primary site approach, and the state machine approach [Jal94]. The state machine approach was chosen for several reasons. The main reason is that the state machine approach more readily takes advantage of having a multicast transport. Also, the state machine approach does not have a single point of failure as does the primary site approach and is simpler to implement and test than most practical voting methods. The main tradeoff is that in contrast to voting methods network partitions in the state machine approach result in disjoint spaces that although coherent and independently functional cannot readily be rejoined.

Multiple Tiered Interfaces

The NCS solution to the classic problem of simplicity versus flexibility is the classic solution of multiple tiered interfaces. Previous examples of tiered interfaces abound, such as Sun's RPC [Sun94], the x-Kernel [HP91], and even the network layer reference models of OSI [DZ83] and TCP [LCPM85].

1.3 Interfaces

NCS provides multiple tiers of interfaces to enable different levels of use. The single DSS interface proved to be an adequate interface for some applications, but was not ideal for many others [Inf98]. Some applications require a high degree of flexibility while others only require a very simple and easy to use interface. NCS is designed to provide a simple set of flexible low level primitives operating on byte blocks upon which high level interfaces are also provided. The higher level interfaces make usage assumptions to simplify the interface at the expense of flexibility. All the interfaces may be used within an application; in other words no interface is mutually exclusive with another interface.

Core Interface

The core interface represents the most primitive level of interaction with the NCS system. This interface is a one to one mapping to actual interactions with the NCS system. Since many interactions require a message send and a message receive many operations are broken into a send

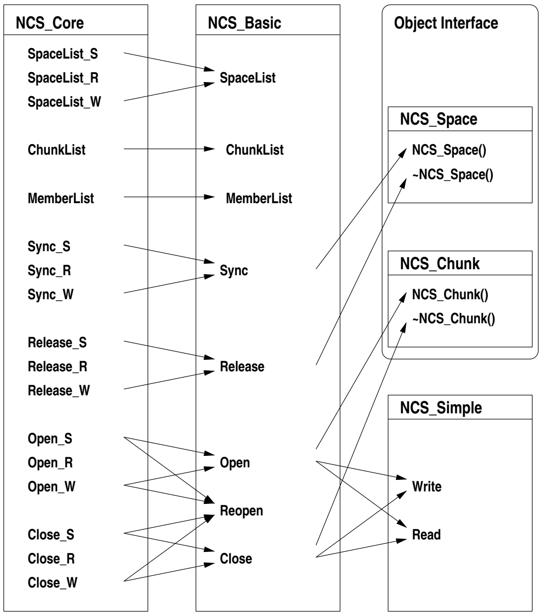

Figure 1: The relationship of some NCS interfaces.

and receive component. This is seen in the existence of calls with _S, _R, and _W suffixes. The _S calls are sends, the _R calls are non-blocking receives, and the _W are blocking receives. The core interface also follows the underlying open/access/close semantics for chunk access. The open/access/close semantics are simply that a chunk is first opened, then accessed, and finally closed. With these semantics a full chunk interaction is not atomic and is therefore inherently more complex for the application developer than atomic access semantics. The core interface is admittedly cumbersome but provides a very high degree of flexibility.

Basic Interface

The main thing this interface does is make the assumption that most interactions with NCS will be blocking calls that do not need to do anything between the core level send and receives. This results in the consolidation of send and blocking receive pairs (core _S and _W calls) into single calls. The open/access/close semantics of the core interface are retained. This interface is often a useful balance between simplicity and flexibility but still carries the complexity of non-atomic chunk interactions.

Simple Interface

The simple interface provides read/write semantics instead of the open/access/close semantics the the lower level interfaces. Read/write semantics are simply that a chunk is read or a chunk is written in a single call. This provides atomic chunk interactions and is therefore simpler for the application developer to use.

Object Interface

Even though the aforementioned interfaces are currently provided as C++ classes, they are more just containers for functions than objected oriented objects. The object interface provides a space class and a chunk class mapping to NCS spaces and NCS chunks in a true object oriented fashion.

DSS Interface

The DSS interface implements DSS_Connection [Inf98] semantics using the core and basic interfaces. The DSS interface features some Linda influenced notions [GCCC85] due to the influence of DSS's predecessor, Harness [Dyn96], which was directly influenced by Linda. This includes the notions of tuples, tuple spaces, and in/out semantics. The influence of Linda can also be seen in other library implemented distributed shared memory systems including of recent note Sun's JavaSpaces within Jini [Sun99]. See DSS for more details on the DSS interface itself [Inf98].

2 Implementation

2.1 State Machine Approach

NCS is implements what is often called the state machine approach to provide data replication [Jal94] [Sch90]. The basic idea of the approach is that there is an identical state machine maintained on each participating node. This state machine services any request for chunk operations. Since every state machine is identical, any server can service any request. However, NCS provides an exception to this rule in that the data itself is not considered part of the state machine's state, only the location(s) of the data. Therefore, in NCS data need not be stored in every participating server, but in k+1 servers where k is the number of node failures that must be tolerated.

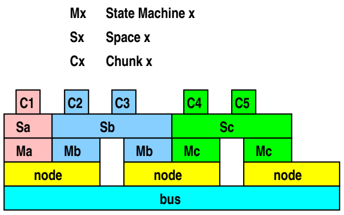

Figure 2: Nodes, state machines, spaces, and chunks.

In Fig. 2 a scenario is shown in which there is a cluster of three nodes each supporting at least one state machine. The figure also introduces the concepts of spaces and chunks which are described below from an NCS implementation point of view. These same concepts are also described in the usage section from an application developer's point of view.

Chunk

A chunk is a named byte block. The name itself is a string. The byte block may be of arbitrary size including a size of zero. The byte block, or data, may be replicated among some number of nodes depending on the chunk's configuration.

Space

A space is a named collection of identical and synchronized state machine instances. In other words, a space is merely the content of a state machine.

State Machine

The state machines instances for a particular space must always be consistent with one another. The ordered multicast layer delivers all messages to all nodes, and hence all state machines, in exactly the same order. State machines are only changed through messages delivered by the ordered multicast layer. Therefore, once state machines are synchronized they will remain consistent with each other.

The chunk state machines for a space maintain a table of all chunks that exist in the space. The state for each chunk exists as a request queue, and the state of the memory itself. The state of the memory exists as a current member node list and a pending member node list. The data itself is not part of the state in that the data only exists at nodes which are on the current member node list, which is not necessarily all nodes with a synchronized state machine instance. State transitions correspond to the two main chunk operations, open and close.

Node

A node is a participating member of a bus. In practice a node is typically an instance of NCS_Bus. A node is uniquely identified by a multicast IP address and a multicast group membership id. The multicast group membership id is a one to one mapping (currently the exact same number) to the underlying reliable multicast implementation id. Therefore, there may be more than one node per machine per multicast address.

Bus

The bus is the reliable multicast backplane which delivers messages to the state machine in a globally ordered manner. This means that every state machine will receive every message in exactly the same order. This is known as *stable* delivery [Jal94] [Sch90].

2.2 Remote Access

The previously discussed state machine approach as described provides for servers. These servers may be embedded in application process to provide an application with NCS functionality as shown in Fig 3. The figure shows an application using space *Sb*.

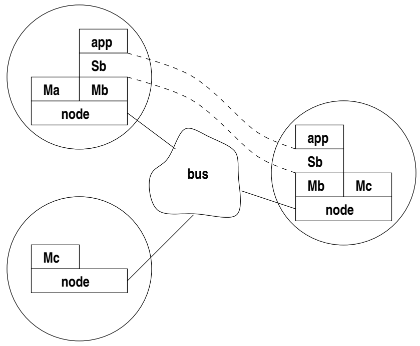

Figure 3: A cluster of servers corresponding to Fig. 2

In many applications this may not be appropriate. It is often better to have an application access NCS in a client server fashion as shown in Fig 4.

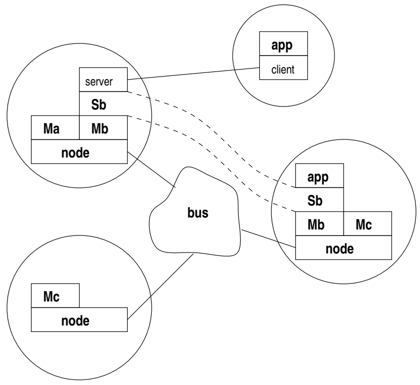

Figure 4: The same cluster as in Fig. 3 but with a remote client.

In this figure some of the application code is moved off into a separate process which connects to NCS through some client code which then accesses NCS. This client/server code is usable through the core interface NCS_Remote and is implemented upon TCP.

2.3 The Protocol Stack

As with most communication methods NCS has an underlying protocol stack.

Note that the NCS protocol stack is bifurcated. In Fig. 5 the left side represents the stack for remote access while the right side represents the stack for local access.

Network

This corresponds to the Network layer in the OSI [DZ83] or TCP/IP [LCPM85] models.

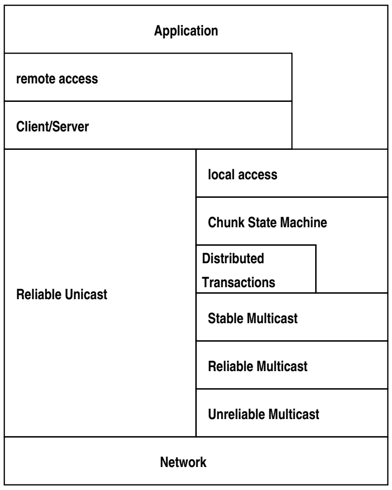

Figure 5: The protocol stack within the NCS system.

Reliable Ordered Unicast

This layer is in support of the traditional client/server aspects of NCS. The requirement for this layer is to provide connection oriented reliable ordered communication. TCP is good at exactly that, so it is used as this layer.

Unreliable Multicast

The requirement for this layer is to provide unreliable packet multicast. Straight IP multicast is used.

Reliable Multicast

The requirement for this layer is to provide multicast service with the following attributes.

- sender ordered packets
- reliable delivery
- fixed packet size

The open sourced RMF/RAMP reliable multicast implementation is used [TAS98].

Stable Multicast

The requirement for this layer is to add the following attributes to the *Reliable Multicast* layer.

- global ordered packets
- variable message size

NCS uses RMF/RAMP in a fixed packet size mode and implements fragmentation and reassembly to provide support for arbitrary length messages. Also, messages are delivered in a way to *stable* delivery of messages, where stable delivery is as defined in [Jal94] [Sch90]. In order for the state machine to work properly, global ordered (not merely sender ordered) message delivery must be provided. This means that all nodes receive all messages in exactly the same order. This is similar to the virtual synchrony of ISIS [BJ87].

Distributed Transactions

Some aspects of NCS functionality such as state machine startup and space list queries require a transaction mechanism. Nested transactions are not required, although concurrent transactions are required. This layer provides a simple commit/abort transaction monitor service.

Chunk State Machine

This layer maintains the state of the memory chunks in NCS. A state machine corresponds to an NCS space.

Local Access Layer

This is the base level interface layer to NCS. The class NCS_Local implements a core interface as this layer.

Client/Server

This layer provides client server semantics for the remote access layer. The Coral Library DNA is used.

Remote Access Layer

This is a semantically equivalent layer to the local access layer. However, instead of directly accessing the state machine, the remote access layer uses the local access layer through the client/server layer. The class NCS_Remote implements a core interface as this layer.

Application Layer

In addition to application code this layer includes higher level interfaces provided by NCS such as NCS_Basic, NCS_Simple, and NCS_DSS_Connection.

2.4 Architecture Overview

This section describes some implementation details that may be helpful when debugging application using NCS application or in advanced usage of NCS.

Member and Pending Lists

For each chunk two lists of nodes are maintained; the member list and the pending list. A member node is a node that has an up to date copy of the chunk memory. A pending node is a node that will make a copy of the chunk memory on the next update.

The order of nodes in the member list are important. The first node in the list and is considered the *alpha* node. In situations where a single node and only a single node must do a chunk specific operation is it the *alpha* node that does it (such as broadcasting a memory update). This is an implicit form of centralized serialization that does not have a single point of failure. If the *alpha* node fails the next node on this list is automatically the *alpha* node upon detection of the failure.

Bus Communication

Communication on the bus is made up of the following message types:

NCS_DATA Node level data.

NCS_PING Requests an NCS_NULL from every other node.

NCS_NULL Out of band empty message.

NCS_MACHINE Machine level messages.

NCS_HBEAT In band empty message.

NCS_TRANSACT_S Transaction send message.

NCS_TRANSACT_R Transaction reply message.

NCS_DEATH_SEED Message to terminate a particular node.

NCS_LEAVE Message to indicate a node is leaving.

Technically, NCS_PING, NCS_NULL, and NCS_DEATH_SEED are all out of band in that order of delivery is not guaranteed to be in exactly identical order on all nodes. NCS_HBEAT is guaranteed to be in exact order so it is used to keep the logical clock incrementing in low usage periods.

The NCS_MACHINE messages are further broken down into the following types:

NCS_M_ADD_MEMBER Add a new member to the state machine.

NCS_M_RM_MEMBER Remove a member from the state machine.

NCS_M_ADD_CLIENT Add a client (chunk operation). This corresponds to an open operation.

NCS_M_RM_CLIENT Remove a client (chunk operation). This corresponds to a close operation.

NCS_M_ADD_PENDING_MEM Add a node to the pending member list.

NCS_M_BROADCAST_MEM This is a request to the *alpha* node to broadcast a chunk memory update.

NCS_M_UPDATE_MEM This is a chunk memory update. This message contains the actual chunk data.

Operations

The definition of an *operation* for the purposes of NCS is any *application* interaction with the NCS system. An operation may consist of a sequence of multiple interface calls (see Actions).

Here are the operations:

SpaceList Get a list of all spaces the bus.

ChunkList Get a list of all chunks in a space.

MemberList Get a list of all nodes synced to a space.

Sync Synchronize a node to a space.

Release Release a node from a space.

Open Open access to a chunk.

Close Close access to a chunk.

Actions

Many operations may be executed with two calls, a send call and a receive call (see NCS_Core). The sends and receives many be interleaved so it is necessary for the receive calls to have a way to specify the proper matching operation. This is done through actions.

When a send is called it initiates some operation in NCS. An *action* is created corresponding to the operation. An *action id* is returned specifying the action. The receive call uses the action_id to specify which action, and hence which send, it corresponds to.

Here is an example where *core* is an interface class derived from NCS_Core.

```
NCS_ActionID action_id;
action_id = core.Sync_S("spacename");
core.Sync_R(action_id);
/* Use space for a while */
action_id = core.Release_S("spacename");
core.Release_R(action_id);
```

Note that in typical applications this level of detailed control is not necessary and the NCS_Basic interface may be used which hides actions altogether as in this example.

```
basic.Sync("spacename");
/* Use space for a while */
basic.Release("spacename");
```

Internally, rendezvous on an action may be through blocking or non-blocking checks. In the case of a blocking check, the check does not return until the status of the action is in some final state (state > NCS_ACTION_RELEASED). In the non-blocking case the check returns even if the state of the action is not in a final state. From the core interface (such as NCS_Core) point of view, blocking action checks correspond to the *_W calls (such as Open_W) while non-blocking checks corresponds to the *_R calls (such as Open_R).

Request Queues

In the state machine a queue of chunk access requests is maintained. A chunk access request is simply the result of an open operation. The requesting node is the node from which the request is made. The local node for a state machine instance is the node that the instance is on. The behavior of a request is controlled by the flags associated with it. The following user flags are available:

NCS_READ Open chunk for reading.

NCS_WRITE Open chunk for writing.

NCS_BLOCK Block the opening of the chunk for reading until memory is valid

NCS_INVALIDATE When the chunk is closed it will be invalidated.

The following flags are set by the system:

NCS_LOCAL The requesting node is the local node.

NCS_OPEN The open chunk operation has returned.

NCS_CLOSE The close chunk operation has returned.

Note that the requesting node for a particular request is the same for all state machine instances while the local node is different for all state machine instances. The 'state' of node locality is therefore not really overall state machine 'state' but is just local data to each instance. This means that different state machine instances will have different NCS_LOCAL flag settings for the same access request.

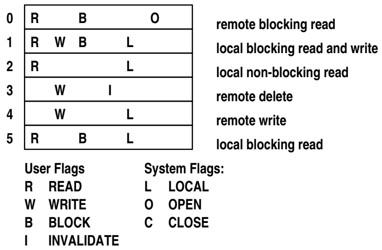

Figure 6: A sample request queue with 5 requests.

The request queue is processed whenever the queue state changes, typically through operations. The basic idea behind the processing of the queue is to provide single writer multiple reader access. This means that at any given time there may be one writer or one or more readers or no access at all. There may not be a writer and a reader accessing at the same time except for the case of a single writer that also has read access. The following algorithm is used to process the queue.

```
if NCS_READ and NCS_WRITE flags set
    if there is existing data
        if requesting node is on memberlist
            open with memory access
        else
            add node to pending node list
            send NCS_M_UPDATE_MEM
    else
        if NCS_BLOCK flag set
            if there is a waiting NCS_WRITE operation move it to top of queue
        else
            open with memory access

else if NCS_WRITE flags set
    open with memory access

else if NCS_READ flag set
    if there is existing data
        for all NCS_READ operations at head of queue
            if requesting node is on memberlist
                open with memory access
            else
                add node to pending node list
                send NCS_M_UPDATE_MEM
    else
        if NCS_BLOCK flag is set
            if there is a waiting NCS_WRITE operation move it to top of queue
        else
            open without access (wouldblock)
```

Typically a new request is added to the end of the queue. However, it is possible to insert a new request into the queue directly after another request. This is done by specifying the descriptor of the other request in the Open operation. However, the new operation will not be inserted before any other operation that is opened (NCS_OPEN flag set; not necessarily an *Open* operation request) and will be inserted after the last opened operation. Note that a request cannot be inserted at the head of a non-empty queue since this would almost always cause corruptive behavior.

One possible use for mid queue insertion is to effectively provide chunk resizing. For example, using an NCS_Basic interface *basic* and NCS_Core derived interface *core*:

```
NCS_Config c;
NCS_Descriptor d;
NCS_Memory *mem;
basic.Open("space", "chunk", NCS_READ|NCS_WRITE, 100, d, &mem, c);
/* We discover that 1000 bytes is needed which is more than 100 and we do    */
/* not want pending NCS_READ operations to happen before we fix it.          */

/* Send insert request */
NCS_ActionID action_id = core.Open_S("space", "chunk", NCS_READ|NCS_WRITE,
    c.replicationLevel, c.memoryType, 1000, NCS_SIZE_MATCH, d);

/* Close old request */
basic.Close("space", "chunk", d);

/* Receive insert request */
core.Open_R(action_id, d, &mem);

/* Use larger memory */

/* Close new request */
basic.Close("space", "chunk", d);
```

The NCS_Basic convenience function *Reopen* is implemented in this way, and is an easy way to resize.

3 Usage

3.1 Space Management

What is a space?

A space is the following:

- state machine for chunk management
- name space for chunks
- association of nodes
- entry points
- replication domain

Space Operations

Synchronization In order to use NCS from a particular node, that particular node must be added to a space. A node is added to a space by synchronizing a chunk state machine with chunk state machines on other nodes for the space. Typically this is done through a *Sync* interface call (such as NCS_Core::Sync() or NCS_Basic::Sync()).

Release A node leaves a space by releasing its chunk state machine from the other state machines for the space. Typically this is done through a *Release* interface call (such as NCS_Core::Release() or NCS_Basic::Release()).

Space Usage Issues

Performance Note Synchronizing a node into a space is a relatively expensive operation with a component that increases linearly with the number of chunks because the state machine's state itself must be transferred to the new node.

This is a one time cost for each synchronizing node, but it may need to be a design factor in environments with real time considerations. However, the latency of the synchronization is somewhat hidden to pre existing nodes since subsequent state machine requests are queued up on the new node until the state is transferred. This means existing nodes need not block until the new node is synced.

Limits A space cannot exist without nodes, therefore it must have at least one node. It is not an error for the last node of a space to *Release* the space; the space will simply cease to exist.

Any chunk within a space can only be replicated as many times as there are nodes to have replicas.

A single space cannot be made up of *nodes* from more than one *bus*. Therefore, the maximum size in *nodes* that a space can be is the number of *nodes* on the bus.

3.2 Chunk Management

What is a chunk?

A chunk is a named byte block. Chunks exist within the namespace provided by spaces.

Chunk Operations

Open To access a chunk it must be opened. The open operation is controlled through four flags.

NCS_READ Open chunk for reading.

NCS_WRITE Open chunk for writing.

NCS_BLOCK Block the opening of the chunk for reading until memory is valid

NCS_INVALIDATE When the chunk is closed it will be invalidated.

Combinations of flags are allowed and some of the common cases are:

non-blocking reader
NCS_READ

blocking reader
NCS_READ |NCS_BLOCK

non-blocking reader/writer
NCS_READ |NCS_WRITE

blocking reader/writer
NCS_READ |NCS_WRITE |NCS_BLOCK

writer
NCS_WRITE

deleter
NCS_WRITE |NCS_INVALIDATE

consumer
NCS_READ |NCS_WRITE |NCS_BLOCK |NCS_INVALIDATE

Note that NCS_BLOCK only has an effect if NCS_READ is set. A pure writer will not block to wait for valid data.

Close An operation stays in effect until it is closed. An operation is automatically closed if the node from which the operation originated releases the space (See Space Management). The space is automatically released from a node is that node dies unexpectedly.

Write The NCS_Simple interface provides read/write semantics which differs from open/close semantics in that the operations are atomic from the application developer's point of view. The write operation simply writes data to a chunk.

Read The read operation simply reads data from a chunk. Note that since this is an atomic operation the internally allocated data from the operation must be freed in the application or else a memory leak may occur.

Chunk Usage Issues

Request Queue Open operations for any particular chunk are handled as chunk access requests in a request queue for that chunk. In general these requests are handled in first in first out order with single writer multiple reader access. The primary exception to first in first out order is that if there are reader(s) (requests with NCS_READ set and NCS_WRITE not set) waiting at the head of the queue, there is no valid data written yet, and there is a writer (request with NCS_WRITE set) later in the queue, the writer is handled before the readers. Otherwise, the readers would necessarily block forever.

The request queue is covered in more detail in the design section.

Blocking Note that there are two distinct types of blocking when accessing chunks.

NCS_BLOCK flags blocking If a reader (Open operation with both NCS_BLOCK and NCS_READ set) gets to the head of the queue and there is no valid data currently available it will block. The operation will unblock as soon as a writer (Open operation with NCS_WRITE set and NCS_READ not set) is added.

Interface blocking This is also known as action based blocking (See Actions).

A 'read' core interface call (*_R calls in NCS_Core) will not block. A 'wait' core interface call (*_W calls in NCS_Core) will block until the operation completes.

Note that a returned flag based non-blocking (NCS_BLOCK not set) that would have blocked is considered a complete operation from the interface point of view. In such a case NCS_RC_WOULDBLOCK is the return code.

Example blocking cases These cases assume no failure and proper usage (valid chunknames, valid spacenames, etc.)

flag settings

	NCS_READ	NCS_READ|NCS_BLOCK
Open_R	**Would Block Case** call does not block operation completes return NCS_RC_WOULDBLOCK	**Pending Case** call does not block operation does not complete return NCS_RC_PENDING
Open_W	**Would Block Case** call blocks until request reaches head and operation completes return NCS_RC_WOULDBLOCK	**Block For Valid Data Case** call blocks until request reaches head and operation completes return NCS_RC_OK

Figure 7: Some example blocking case for when Open_S has been called on a new chunk.

Pending Case Suppose we make an Open_S call with NCS_READ|NCS_BLOCK set on a new chunk. Therefore, there is no previous valid data written. Then we do a Open_R. Open_R will not block and will return with the return code NCS_RC_PENDING.

The operation is not complete and Open_R (or Open_W) may be called again.

Block For Valid Data Case Suppose we make an Open_S call with NCS_READ|NCS_BLOCK set on a new chunk. Therefore, there is no previous valid data written. Then we do a Open_W. Open_W will block until data is written. When Open_W will return with the return code NCS_RC_OK.

The operation is then complete and Open_R or Open_W may not be called again without another Open_S call.

WouldBlock Case Suppose we make an Open_S call with NCS_READ set on a new chunk. Therefore, there is no previous valid data written. Then we do a Open_R. Open_R will not block and will return with the return code NCS_RC_WOULDBLOCK. Suppose we do an Open_W instead of the Open_R. The call will block only until the operation reaches the head of the queue at which time it will return NCS_RC_WOULDBLOCK.

The operation is complete because the chunk request for it got to the head of the queue and was resolved as NCS_RC_WOULDBLOCK since the NCS_BLOCK flag was not set.

Read Works Case Suppose we make an Open_S call with NCS_READ set on a previously written chunk. Therefore, there is valid data available. Then we do a Open_R. Open_R will not block and will return with the return code NCS_RC_OK.

Suppose we do a Open_W instead. Open_W will not block and will return with the return code NCS_RC_OK.

The operation is complete with either Open_R or Open_W.

Since data was available all settings behave the same. If NCS_BLOCK was set for the Open_S the behavior would still be the same.

Basic Interface Case The *basic* NCS interface (NCS_Basic) implements its Open() call with an Open_S followed by a Open_W call. Therefore, a NCS_Basic::Open() call with NCS_READ|NCS_BLOCK set will block until valid data is available. A NCS_Basic::Open() with NCS_READ set and NCS_BLOCK not set will return NCS_RC_OK is there is valid data and NCS_RC_WOULDBLOCK is there is not.

Note that NCS_RC_PENDING should never be returned by NCS_Basic::Open() because the underlying Open_W will always wait for the request to get to the head of the queue.

Blocked on Writer Case Suppose we make an Open_S call with NCS_READ on a chunk being written by a slow writer. Therefore, there is a request with NCS_WRITE set at the head of the queue. The we do a Open_W. Open_W will block until the writer is done. Note that even though NCS_BLOCK was not set this call still blocks.

If we has done a Open_R instead it would have returned NCS_RC_PENDING.

3.3 Replication

Chunks may have multiple copies of data among the nodes of a space. This is called replication.

Modes

There are three basic modes in which replication can be set. These modes are set on a chunk by chunk basis, so different chunks may replicate in different modes.

Fixed Mode The replication in fixed mode is set to some exact number. This number is the *replication level.* At replication level R, the number of copies of the data NCS will try to maintain is R+1; the original copy plus R 'replicas'.

Full Mode In full replication mode NCS will maintain a copy on every node in the space. Therefore, if there are M nodes the replication level R will be maintained at M-1.

Passive Mode In passive replication mode NCS will not actively adjust the number of replicas (extra copies of data). If a node needs a copy of data and does not currently have one a replica will be made on that node and the replication level will have passively incremented by one. If a node with a replica leaves the space then the replication will have passively decremented by one.

Replication mode is set through a configuration field that has the following possible values:

NCS_NONE fixed replication mode, level 0

NCS_FULL full replication mode

NCS_SAME do not change replication mode

NCS_PASSIVE passive replication mode

some integer X fixed replication mode, level X

NCS_DEFAULT NCS_NONE is the default

Under Replication

In fixed mode there will be less replicas than is set if there are not enough nodes in the space to support the replication level. This is called under replication and happens for replication level R and number of nodes M if $M < R+1$.

This is automatically corrected for (replicas are made) as nodes are added to the space.

Over Replication

Replication adjustments are done at write time. Therefore, if a node without a replica does a read a new replica will be made at that node. The effective replication level is now higher than the specified level if in fixed mode. This is called over replication. This is a caching effect and is often desirable. The replication will be corrected when a write is done.

Fault Tolerance

Data fault tolerance is achieved through data replication. As long as there is a node with a copy of the data then the data is not lost. Therefore, at replication level R, the data will not be lost with R node failures since there will be R+1 copies.

Cache Coherency Point of View

Distributed access to chunks can be considered a cache coherency problem [Hwa93]. The different replication modes provide different cache coherency solutions, thereby providing the application with the ability to choose the most appropriate one for its needs.

Write Invalidate One common cache coherency method is called write invalidate. Write invalidate means that when data is changed (written) at a particular node, all other cache copies on other nodes are invalidated. This corresponds to fixed mode replication level 0, or the NCS_NONE replication level setting.

Write Update Another common cache coherency method is called write update. Write update means that when data is changed (written) at a particular node, all other cache copies on other nodes are updated to the new data. This corresponds to passive mode replication.

Latency Issues Latency for a large data operation in NCS is usually most influenced by the copying of data from node to node. This latency happens either at write time or read time depending on replication mode.

In fixed mode the latency for the fixed level of copies is at write time while the latency for a read on a node without a copy is at read time. In fixed mode level 0 (the NCS_NONE replication setting) the latency is always at read time.

In full mode the latency is always at write time.

In passive mode the latency is at 'first' read time, and write time otherwise. The first read from a node without a copy will have to get a copy. Once a node has a copy writes will do copies to that node.

3.4 Architecting with NCS

Due to the tiered interfaces of NCS and the modular design of the primary NCS components, architecting an NCS application has many options.

Building Blocks

Bus This the communication backplane upon which all internal NCS communication is done. This communication backplane exists as a collection of *nodes* which communicate with each other through reliable multicast. If no *node* exists then there is no *bus*.

Since the bus is a one to one mapping to the underlying multicast service every bus needs to have its own multicast address. This is important to keep in mind for both architecture design and implementation. During design, if multicast addresses are considered a limited resource, the number of buses are a factor. During implementation, the implementor must be careful to explicitly set multicast addresses (that does not mean hard code them) for the buses since the default address is the same for any instance.

Node A node is an object which may send and receive messages on a *bus*. The class NCS_Bus is the most common type of *node*. *Nodes* are identified by a unique unsigned 32 bit integer.

Space A *space* is a named namespace for *chunks*. A *space* exists as a collection of one or more state machines on *nodes*. In order to access a *space* from a particular *node*, a state machine for the *space* must be **synchronized** on that *node*. Since a *space* exists as a collection of synchronized state machines, if there is no synchronized state machine there is no *space*.

A particular *space* may only exist among *nodes* of the same *bus*. In other words, a single *space* cannot exist across multiple *buses*. If a two state machines are synchronized with the same name on *nodes* of different *buses*, they are not part of the same *space*. They are parts of separate *spaces* that happen to have the same name.

Chunk A *chunk* is a named byte block. The whole point of NCS is to be able to easily and efficiently read and write *chunks* in a distributed manner.

Core Interface The *core* interface is specified by the class NCS_Core. NCS_Core is an abstract class which defines the base interface of primitives to the NCS system. NCS_Core is the foundation for all other interfaces to NCS.

Local Interface The most common *local* interface is NCS_Local. NCS_Local is a class derived from NCS_Core which is associated directly with a *node* (the constructor even requires an NCS_Bus argument).

Remote Interface The most common *remote* interface is NCS_Remote. NCS_Remote is derived from both NCS_Core and DNA_Client and requires a *server* with which to interact.

Server A *server* in NCS terminology is an object that handles interaction from a *remote* interface by directing it to a *node*. The most common server is NCS_Server which is associated directly with an NCS_Core derived interface.

Application An *application* in NCS terminology is code which uses an NCS interface.

Basic Interface The basic interface is a higher level interface than the *core* interface. The basic interface is implemented by NCS_Basic. Most end NCS usage is not done with a lower level interface than NCS_Basic. NCS_Basic is simpler and easier to use than the *core* interface, but sacrifices some fine grained flexibility. The NCS_Basic interface is completely intermixable with the NCS_Core interface.

For top level system architecting purposes the *basic* interface can be considered part of the *application*.

DSS Interface The DSS interface is implemented by NCS_DSS_Connection and is a drop in replacement for DSS_Connection. This provides a tested interface that is a little higher level than NCS_Basic. See DSS_Connection manpage [Inf98] for details.

For top level system architecting purposes the DSS interface can be considered part of the *application*.

Space and Chunk Oriented Interface The other interfaces tend to be 'point of entry' oriented objects. In other words the interface objects tend to map to *nodes*, *servers*, or connections (such as is the case with DSS).

Another approach is to orient the objects towards *spaces* and *chunks*. With this interface the class NCS_Space corresponds to a *space* and NCS_Chunk corresponds to a *chunk*.

For top level system architecting purposes this interface can be considered part of the *application*.

Using the Blocks

There are several common entities that can be assembled with the available building blocks. In order to more clearly portray assemblies of building blocks the following notation will be used:

N node

S server

L local interface

R remote interface

A application code

Some of the possible entities are summarized in the following sections. The usual way of implementing these entities are as traditional processes in the UNIX or NT sense.

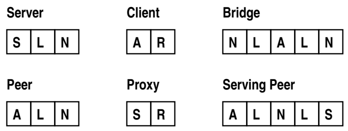

Figure 8: Some Common Entities

Traditional Server (SLN) This is the simple case of using a *server* with its required *node*. This entity can asynchronously serve multiple clients.

Traditional Client (AR) This is the simple case of an *application* using a *remote* interface. As would be expected this entity requires the existence of a *server*, such as an SN server, to function.

Peer (ALN) When an *application* directly uses a *local* interface and the *local*'s required *node* the end result is a participant in a peer to peer environment.

Serving Peer (ALNLS) By adding a *server* to the plain ALN peer it is now possible for the peer to not only communicate with other peers but to serve clients as well.

Bridge (NLALN) An application may communicate across two or more buses through this entity.

Proxy (SR) This server connects to another server and can be used to relay services across mediums that cannot support the underlying IP multicast.

Some Example Architectures

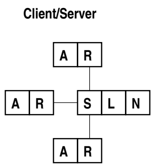

Figure 9: Client/Server Example

Client/Server The traditional notion of a simple client/server architecture is to have a centralized *SN server* with one or more *AR clients*.

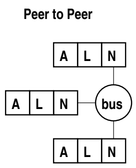

Figure 10: Peer to Peer Example

Peer to Peer A basic peer to peer architecture is simply a collection of *ALN peers*.

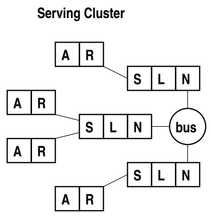

Figure 11: Serving Cluster Example

Serving Cluster By having multiple client/server clusters connect through a bus a serving cluster results. (This is the basic architecture for DSS).

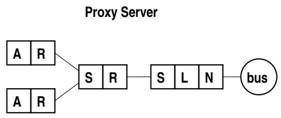

Figure 12: Proxy Server Example

Proxy Server An example with two clients using NCS through a proxy server.

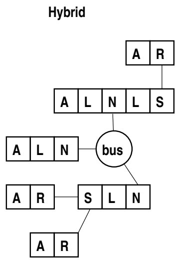

Figure 13: Hybrid Example

Hybrid Many variants are possible by mixing up the components.

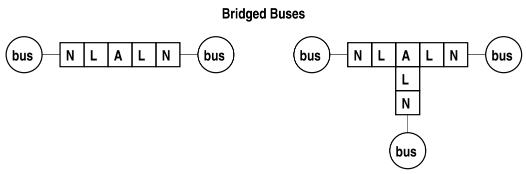

Figure 14: Bridged Buses Example

Bridged Buses This is simply the application of a *NLALN bridge*. Note that a *bridge* is not restricted to two buses.

Multiple Buses

Multiple Spaces

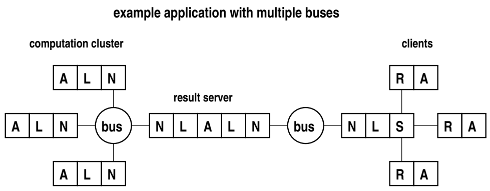

Figure 15: Multiple Buses Example

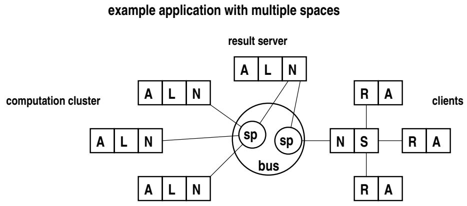

Figure 16: Multiple Spaces Example

Bus Performance Configuration

The bus itself has several configuration parameters that affect performance. These parameters deal with internal mechanisms and are as follows:

blockingRecv If TRUE use the reliable multicast implementation's blocking receive mode. This can slightly reduce response latency but since RAMP's blocking mode is implemented as a spin loop the machine may be loaded more.

sendSpacer The time in milliseconds between checks for queued out of band messages to be sent. Since most important messages are in-band this parameter has little impact.

recvSpacer The time in milliseconds between checks for incoming messages if blockingRecv is FALSE.

heartBeatTime The time in milliseconds between heartbeats.

heartBeatsToCheck Number of heartbeats between garbage collection passes. This primarily affects how often hosts are checked to be dead.

consideredDeadTime The time in milliseconds a host must not respond in order to be considered dead.

stabMode The message stabilization mode.

The parameter stabMode may have one or more of the following settings:

NCS_HEARTBEAT Heartbeats are sent every heartBeatTime milliseconds. This helps keep inactive nodes from being considered dead.

NCS_ACTIVE_ECHO A dummy response is sent every time a message is received. This corresponds to an ACK based protocol and can reduce latency. However, it is not very scalable.

NCS_AVOID_BEING_LAST Nodes check to see if they are the last node to have sent a non trivial message. If so a node sends a heartbeat. This more scalable type of ACK protocol with slightly higher CPU overhead than NCS_ACTIVE_ECHO.

NCS_LAN_ONLY This makes some assumptions to help performance when all nodes are in the same collision domain. Do not use this with switches.

In C++ the parameters are adjusted with NCS_Bus::SetConfig(NCS_BusConfig &config) although there are three preset modes that may be set with NCS_BusConfig::UsePreset(NCS_BusConfigMode configmode). These modes are:

NCS_MODERATE The default. Reasonable values for average usage.

NCS_LOW_IMPACT Minimizes overhead at the expense of responsiveness.

NCS_FAST_RESPONSE Minimizes response times at the expense of overhead.

3.5 Examples

Writer example

```
#include <ncs/ncs.h>

#ifdef USE_CORE

/***************************************************************************
This first example uses the NCS_Remote interface directly.  NCS_Remote is
derived from NCS_Core and is used to do NCS operations through a remote
server.  By default NCS_Remote will attempt to connect to a server on
localhost on port ncs_server (in /etc/services).
***************************************************************************/
int main(const int argc, const char *argv[])
{
    NCS_Remote core;
    NCS_ActionID action_id;
    NCS_ReturnCode rc;
    NCS_Descriptor d;
    NCS_Memory *memory;

    core.Initialize();

    action_id = core.Open_S(argv[1], argv[2], NCS_WRITE, NCS_FULL,
        NCS_SAMETYPE, strlen(argv[3])+1, NCS_SIZE_MATCH, 0);

    rc = core.Open_W(action_id, d, &memory);

    if(rc == NCS_RC_OK && memory->GetPointer() != NULL)
    {
        osdMemcpy(memory->GetPointer(), (void *)(argv[3]), strlen(argv[3])+1);
    }

    action_id = core.Close_S(argv[1], argv[2], d);
    rc = core.Close_W(action_id);

    return 0;
}

#elif defined USE_BASIC

/***************************************************************************
This example behaves the same as the previous example but uses the simpler
NCS_Basic interface.
***************************************************************************/
int main(const int argc, const char *argv[])
{
```

```
    NCS_Remote core;
    NCS_Basic basic(&core);
    NCS_ReturnCode rc;
    NCS_Descriptor d;
    NCS_Memory *memory;

    core.Initialize();

    rc = basic.Open(argv[1], argv[2], NCS_WRITE, strlen(argv[3])+1, d, &memory);

    if(rc == NCS_RC_OK && memory->GetPointer() != NULL)
    {
        osdMemcpy(memory->GetPointer(), (void *)(argv[3]), strlen(argv[3])+1);
    }

    rc = basic.Close(argv[1], argv[2], d);

    return 0;
}

#else /* EXAMPLE==SIMPLE */

/******************************************************************************
This example behaves the same as the previous example but uses the even
simpler NCS_Simple interface.
******************************************************************************/
int main(const int argc, const char *argv[])
{
    NCS_Remote core;
    NCS_Simple simple(&core);
    NCS_ReturnCode rc;

    core.Initialize();

    simple.Write(argv[1], argv[2], (void *)(argv[3]), strlen(argv[3])+1);

    return 0;
}

#endif
```

Server Example

```
#include <ncs/ncs.h>

int main(const int argc, const char *argv[])
{
    NCS_Bus bus;
    NCS_Local core(&bus);
    NCS_Server server(&core);
    NCS_ActionID action_id;
    NCS_ReturnCode rc;
    GPL_Boolean n;

    bus.Open();
    n = server.Initialize();

    action_id = core.Sync_S(argv[1]);
    rc = core.Sync_W(action_id);

    while(1)
    {
        server.Wait();
    }

    return 0;
}
```

4 Closing Notes

4.1 Availability

NCS as well as the rest of the Coral Tree Library Set is available under the Artistic license which is the same open source license Perl uses.

The Coral Tree Library Set may be obtained from http://www.imonk.com/download.html or ftp://ftp.imonk.com/vendors/weber/download/coralsrc.

4.2 Future Work

Other Reliable Multicast Implementations

Although RMF/RAMP provides a useful implementation of reliable multicast it is desirable to have the ability to use other implementations. Although the current NCS implementation is modular enough to make this relatively easy it has not yet been done.

Port to Win32

Although most of the implementation of NCS is portable to Win32, the current reliable multicast, RMF/RAMP, is not. Either RMF/RAMP must be ported to Win32 or another reliable multicast must be used. Currently, NCS works only on the RMF/RAMP supported platforms which are Linux, Solaris, and IRIX.

Atomic Primitives

The NCS_Simple interface provides for atomic read/write semantics, but the underlying implementation does not yet provide the lower overhead primitives which are possible when the interactions are atomic. For example, NCS_Simple::Read is currently implemented as a NCS_Basic::Open with NCS_READ|NCS_BLOCK set followed by a NCS_Basic::Close. Since an atomic read does not need to change state in order to provide coherency or consistency the underlying communication can be reduced. This is especially true if a local replica exists in which case network communication is not necessary at all. Therefore, NCS_Core::Read and NCS_Core::Write should be implemented.

Multiple Writer Multiple Reader Sharing

Currently, only single writer multiple reader sharing is implemented. Allowing multiple writers in some application cases can improve performance dramatically.

Replica Space Size Limits

Currently, replica generation is bounded only by the replication level settings of the chunks. In some cases available space on machines is an important consideration. Therefore, there should be an optional limit to the space used on any particular node for replicas.

Implement the Object Interface

The object interface is not yet implemented. This needs to be done.

Finish the Backing Memory System

DSS has the ability to have the actual chunk data stored in daemon heap, system shared memory, or on the file system. The shared memory option makes SMP operations more efficient while the file system provides a permanent store. This system has been redesigned for NCS in a more flexible and extensible way but still needs to be actually implemented. However, the infrastructure for this has already been implemented and is available as the Coral module NRM.

References

[BJ87] K. P. Birman and T. Joseph. Exploiting Virtual Synchrony in Distributed Systems. In *Proceedings 11th Sym. on Operating System Principles*, pages 123–138, November 1987.

[CBZ91] J. B. Carter, J. K. Bennett, and W. Zwaenepoel. Implementation and Performance of Munin. In *Proc. 13th ACM Sym. on Operating System Principles*, pages 152–164, 1991.

[CBZ95] J. B. Carter, J. K. Bennett, and W. Zwaenepoel. Techniques for Reducing Consistency Related Communication in Distributed Shared Memory Systems. *ACM Transaction on Computer Systems*, 13(3):205–243, August 1995.

[CDK94] G. Coulouris, J Dollimore, and T. Kindberg. *Distributed Systems Concepts and Design*, pages 517–543. Addison-Wesley Publishing Company, second edition, 1994.

[Dyn96] Dynamics Research Corporation. *Harness Manual*, 1996.

[DZ83] J. D. Day and H. Zimmerman. The OSI Reference Model. In *Proc of the IEEE*, volume 71, pages 1334–1340, December 1983.

[GCCC85] D. Gelernter, N. Carriero, S. Chandran, and S. Chang. Parallel Programming in Linda. In *Proceedings of the International Conference on Parallel Processing*, August 1985.

[GGH91] K. Gharachorloo, A. Gupta, and J. Hennessy. Performance Evaluations of Memory Consistency Models for Shared Memory Multiprocessors. In *Proceedings of the 4th Sym. on Architectural Support for Programming Languages and Operating Systems*, New York, April 1991. ACM.

[HP91] N. C. Hutchinson and L. L. Perterson. The x-kernel: An Architecture for Implementing Network Protocols. *IEEE Tans. Software Eng*, 17:64–76, January 1991.

[HST99] Weiwu Hu, Weisong Shi, and Zhimin Tang. JIAJIA: An SVM System Based on A New Cache Coherence Protocol. *Proceedings of the High Performance Computing and Networking (HPCN'99)*, LNCS 1593:463–472, April 1999.

[Hwa93] K. Hwang. *Advanced Computer Architecture: Parallelism, Scalability, Programmability*, pages 348–357. McGraw-Hill, Inc., 1993.

[IEE90] IEEE. ISO/IEC 9945-1: 1990 (IEEE Std. 1003.1: 1990) Information Technology-Portable Operating System Interface (POSIX)-Part 1: System Application: Program Interface (API) [C Language]. Technical report, IEEE, 1990.

[Inf98] Infinite Monkeys Incorporated, Aloha, Oregon. *Coral Tree Library Set User's Manual*, version 1.2 edition, 1998.

[Jal94] P. Jalote. *Fault Tolerance in Distributed Systems*, pages 257–306. Englewood Cliffs NJ: Prentice Hall, 1994.

[Kar99] Kamran Karimi. Dipc, 1999. available from http://wallybox.cei.net/dipc.

[Lam79] L. Lamport. How to Make a Multiprocessor Computer that Correctly Executes Multiprocess Programs. *IEEE Trans. Comput.*, 28(9):690–691, September 1979.

[LCPM85] B. M. Leiner, R. Cole, J. Postel, and D. Mills. The DARPA Internet Protocol Suite. *IEEE Commun. Magazine*, 23:29–34, March 1985.

[LH89] K. Li and P. Hudak. Memory Coherence in Shared Virtual Memory Systems. *ACM Transactions on Computer Systems*, 7(4):321–359, 1989.

[Pit99] Andrew J. Pitman. pvmsync, 1999. available from http://torch.rowan.edu/ãjp/beowulf/pvmsync.html.

[RBMS97] D. Ridge, D. Becker, P. Merkey, and T. Sterling. Beowulf: Harnessing the Power of Parallelism in a Pile-of-PCs. In *Proceedings, IEEE Aerospace*, 1997.

[Sch90] F. B. Schneider. Implementing Fault-Tolerant Services Using the State Machine Approach: A Tutorial. *ACM Computing Surveys*, 22(4):299–319, December 1990.

[SS93] R. D. Schlichting and F. B. Schneider. Fail Stop Processors: An Approach to Designing Fault-Tolerant Computing Systems. *ACM Transactions on Computer Systems*, 1(3):222–238, August 1993.

[Sun94] Sun Microsystems. *RPC manpages*, 1994.

[Sun99] Sun Microsystems. *JavaSpaces TM Specification*, 1999. available from http://www.sun.com.

[TAS98] TASC, Inc. RMF/RAMP, 1998. Copyright (c) 1997, 1998, available from http://www.tasc.com.