1994

A Generic Software Modeling Framework for Building Heterogeneous Distributed and Parallel Software Systems

William T. O'Connell

George K. Thiruvathukal
Loyola University Chicago, gkt@cs.luc.edu

Thomas W. Christopher

A Generic Software Modelling Framework for Building Heterogeneous Distributed and Parallel Software Systems

William T. O'Connell
AT&T Bell Laboratories
Building IX, Rm. 1B-422
1200 E. Warrenville Rd.
Naperville, IL 60566
wto@ihlpx.att.com

George K. Thiruvathukal
R.R. Donnelley and Sons Company
Technical Center
750 Warrenville Road
Lisle, IL 60532
gkt@disney.donnelley.com

Thomas W. Christopher
Illinois Institute of Technology
Department of Computer Science
10 West Federal Street
Chicago, IL 60616
tc@iitmax.acc.iit.edu

Abstract

Heterogeneous distributed and parallel computing environments are highly dependent on hardware and communication protocols. The result is significant difficulty in software reuse, portability across platforms, interoperability, and an increased overall development effort. A new systems engineering approach is needed for parallel processing systems in heterogeneous environments. The generic modeling framework de-emphasizes platform-specific development while exploiting software reuse (and platform-specific capabilities) with a simple, well defined, and easily integrated set of abstractions providing a high level of heterogeneous interoperability.

Index Terms - Heterogeneous computing. Parallel and distributed environments. Generic modeling framework. Lossless domain mapping. Reusability. Portability. Extensibility. Interoperability. Systems Engineering.

1 Introduction

Heterogeneous distributed and parallel systems are widely gaining popularity in a broad variety of applications [11][19][20][22]. The cooperation and coordination between computers, at distinct locations, greatly enhances the usefulness of each individual computer. It provides the ability to orchestrate and coordinate a wide range of diverse high-performance machines (including parallel machines) for computationally demanding tasks that have different CPU needs [15]. However, distributed control leads to systems that have complex designs. We propose that basic foundations, general techniques, and clear methods are essential to improve our understanding and to deal efficiently with high-performance distributed and parallel systems.

Our focal point is on integrating conventional distributed programming over heterogeneous high-performance systems with parallel processing. We use Object-Oriented Design and Programming (OOD and OOP) to define and implement a distributed computing environment. To show the utility of our approach, we implemented *Distributed Memo* (D-Memo) to provide a shared directory of queues over heterogeneous machines [1]. These virtual shared queues are for user level processes, similar to the approach taken by *Linda* [2][8][12]. The scope of our approach is limited to systems engineering. D-Memo itself is not an object-oriented environment. Object-oriented languages (such as Message Driven Computing and Macrodataflow Array Processing Language) are available as alternative interfaces to our distributed computing environment [1]. D-Memo has been chosen, because it contains a sufficient number of elements to illustrate the problems of heterogeneous computing (HC).

In this paper, we will place greatest emphasis on the definition of a general modelling framework on which we believe HC software systems should be defined. To properly motivate this discussion, we will first discuss the *major issues* that impact heterogeneity. We will take this set of issues and use it to establish *foundations* for constructing high-performance heterogeneous distributed software systems. These foundations will be integrated with the help of object-orientation (without which we believe the implementation of the system would be difficult, if not impossible, to achieve). Finally, we will survey related work and present conclusions.

2 Heterogeneous Computing Issues

The issues faced by heterogeneity are obviously hardware platforms, communication protocols, operating system interface differences, and other distinguishing characteristics (e.g. processor speeds, number of processors, etc...) [15][20]. But, the main problem facing HC is data modeling and interconnection. There are three *important* issues on which we will focus:

- Languages Data Type Inconsistencies
- Networking Protocols
- Incompatible Domains Representations

A serious problem facing HC is that most languages take a liberal view of concrete data types[1] (or scalar types) across heterogeneous platforms. Certainly, such a view facilitates language implementation for a given machine, but this view causes menace for practitioners of heterogeneous computing. Consider integers and floating point number representations. Integers are typically defined to be the word size of an architectural platform. Today, architectures are available with word sizes of 16, 32, 64, and 128 bits (as well as arbitrary bit-vectors). Floating point numbers present an analogous problem. In fact, the problem is slightly more serious, since different precision representations are

1. A concrete data type is the device used by language designers to support scalar data and operations efficiently. This efficiency can usually be obtained under the assumption that the data type can be easily mapped onto a given architecture.

common: single-, double-, and extra-precision, for example. The availability of different word sizes and precisions leads further to the problem of handling exceptional conditions[1]. The problems are well-known by virtually all who do numerical processing. In the context of HC, the number of exceptional conditions explodes combinatorially to render exception handling difficult, if not impossible, to achieve.

Networking adds to the complications of heterogeneity. The compelling issue is the availability of different protocols (and implementations of the same protocol). The typical protocol stack used is TCP/IP because of reliability. But with this reliability, comes an overhead cost. It may be more appropriate to use the UDP protocol with or without reliability built on top of it, especially on a local area network with low bit error rate. Another problem is that not all machines support a connection-oriented transport layer. Such a layer must augment the capabilities of the network layer. Take, for example, the INMOS transputer. The transputer provides up to four communication channels per transputer, which allows networks of transputers to be connected by direct point-to-point connections. The network layer has low bit error rate but must be completely managed by user programs. No connection-oriented transport layer exists.

When considering the goal of HC, applications differentiate between code, algorithms, and data to optimize the matching of the computational tasks to the appropriate machine type (e.g. parallel, pipelined, and special purpose architectures) [11]. This ability allows the application to execute the right code on the most optimal machine. But this choice can lead to different protocols between different machines. This presents serious problems for an application using many different protocols when maintaining a level of interoperability.

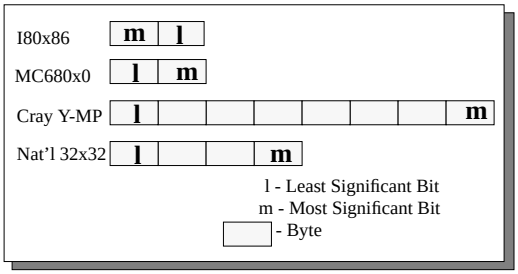


Figure 1 - Integer Representations.

Incompatibility of domain representations, encompassing a superset of the problems associated with the languages mentioned above. Figure 1 illustrates the hardware differences between four selected processors. Not only are the sizes different, but the least and most significant bits are not the same. This results in an incompatible domain representation for the application software. Mapping data types from one machine to another is not transparent.

Typically, networking protocols use what is called the network-byte order when sending information over a network [4]. This still does not resolve the basic issue of transparent data type mapping across machines. The problem is such that it leads to one-way domain compatibility. The domain maps properly to one machine, but not vice versa.

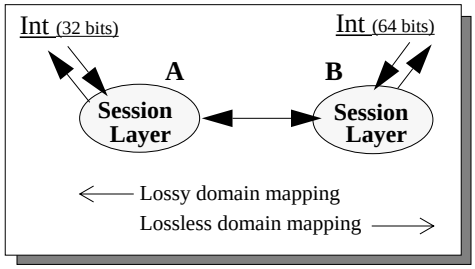


Figure 2 - Lossy Domain Mapping

Figure 2 illustrates the mapping between two distinct architectural implementations of an integer. It is showing the mapping of a 32-bit integer on a machine (**A**) to a 64-bit integer on machine (**B**). This will be a lossless mapping from **A** to **B**, since we will not lose any data (through precision). But, when mapping from machine **B** to **A**, there is a possibility of lost data. This is classified as a lossy mapping [9]. A more concrete example of a lossy mapping is: Mapping an *integer* data value from an Alpha processor (64 bit) to an i860 Processor (32 bit), where the integer value requires more than 32 bits. Thus, representation across heterogeneous machines does not facilitate transparent scalar data mapping.

The above three issues expose the major challenges of HC. Another entirely different set of issues pertaining mainly to parallel processing are operating system interfaces (e.g. locking and synchronization methods, shared memory capabilities, etc..). The previous examples were dealing more with information exchange, while these issues are dealing more with software reuse and porting issues for the application code. Yet there are even more general issues, such as seamlessness, routing, performance considerations, network bandwidth, propagation delays, and efficiency. Therefore, it is not surprising that code reuse is difficult for applications performing tasks over multiple heterogeneous machines.

Using one issue mentioned earlier, synchronization, we will compare the Encore Multimax (with eight processors) with the Sun Sparc 4 workstation (with one processor). In both cases, synchronization primitives are furnished. The Sun utilizes the standard System V operating system primitives. The semaphores must be created in the operating system kernel before a process may use them. Once they exist in the kernel, any process may attach to them. The good news is that the Encore also supports System V semaphores resulting in total synchronization code reuse. The bad news is that the Encore provides more efficient synchronization primitives which are more attractive for performance reasons. Encore offers primitives for semaphores, spin locks, and barriers: Each with different system interfaces, including spin locking or process blocking. These Encore interfaces can be emulated with System V primitives but, will not result in optimal performance. The differences between just two machines are so radical, that when you start considering many platforms, the reuse becomes difficult.

1. Such conditions include inconsistent type conversions and heterogeneity data coercion.

Engineering high quality software on a network of heterogeneous platforms in a general manner is complex and difficult. On one hand, the above discussion suggests that HC is fraught with excessive, not easily managed differences. On the other hand, we have a paradigm, object-orientation, which can be instrumental in managing differences. Discovering commonality is a central tenet of object-oriented design. This commonality is the basis for the foundation building blocks, the subject to which we now turn our attention.

3 Basic Foundation Building Blocks

Because of the complexity in HC, we are proposing a set of generic software modeling techniques through the utilization of frameworks. Each framework is a basic structure (object) which encapsulates a common abstraction for providing a generic interface. A common abstraction (framework) will actually consist of one or more layers, similar to the OSI model.

The generic methodologies of Conway [7] and Kritzinger [16] are among the few published works appearing in literature on generic modeling and performance evaluation of networked systems. Both papers concentrate on the multilayered protocol of the OSI communication architecture. Their methodologies can be combined to generically model the distributed architecture. Combined with object-orientation, we will abstract a generic modeling framework to satisfy the objectives presented in the introductory section.

There has been little interaction between researchers in the areas of HC and object-oriented computing, despite the same terminology being used by both sets of people (most notably, the use of *objects*). Some recent papers suggest that the two areas may benefit from closer interaction [10][17][19][20][22]. Distributed and parallel systems have evolved in response to usable concurrent CPU cycles over one or more machines. Object-oriented computing has evolved for engineering large software systems. We contend that HC systems present a grand software engineering challenge, a challenge which is much better approached with object-oriented methodology than other methods (e.g. functional decomposition or structured analysis). Empirical studies on the relationship between the use of the object-oriented (OO) paradigm and software reuse have shown that the OO paradigm substantially improves productivity and reusability over procedural approaches. [17].

3.1 Analysis of Decomposition

Before decomposing the frameworks (abstractions), we will describe the criteria used in analyzing the components. To provide a combined framework model, the components that make up the model must meet the following requirements:

- Each decomposition is an abstraction that provides services to it's clients[1]. The requests must be generic; which are architectural and protocol independent.
- The model must provide a set of internally consistent abstractions in the context of the larger design. This design is intended to provide a consistent model for high-performance distributed and parallel computing.
- Each framework will consist of one or more abstractions forming a *cluster*, each providing a unified abstraction[2].
- Each abstraction must be well defined and of high quality. The model should be characterized by a set of easy to integrate and efficient abstractions.

By analyzing the **basic** issues associated with HC, we can classify the issues in one of four areas: distributed communication, locking/synchronization methods, shared memory capabilities, and dynamic data migration. With this in mind, we devised a set of generic frameworks. Each of these frameworks contain related issues within its scope. Using these basic abstractions, we can build any system that is **transparent** to the underlying platforms and protocols.

3.2 Generic Modeling Framework.

The generic model provides a framework for developing parallel HC environments. This environment is characterized by four major clusters of abstractions: communication, transferable, shared memory, and locking (see figure 3).

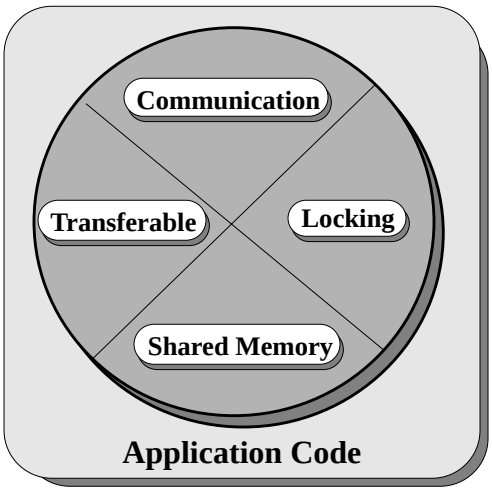


Figure 3 - Generic Modeling Framework

To support a new computer in a heterogeneous network, one must consider each of these four pieces of the pie. Our view is that a new computer can be supported by learning the differences from the base definition (called a base class in object-oriented terminology) and either extending or overriding the services provided in the base definition. An application that uses the services of these core abstractions can reap the benefits of transparency, reuse, portability, and interoperability.

We now turn to a discussion of each of the four frameworks of heterogeneous computing.

1. This is one definition of client-server model.

2. The cluster, or class category, is available in many of the object-oriented methodologies and is a useful analysis technique when the relationship between the classes in the cluster is not completely understood.

3.2.1 Transferable Framework

The transferable framework is the most complicated of the four frameworks. To facilitate understanding of transferrables, we will first motivate the problem of domain incompatibility. Second, we will discuss how to support robust concrete types with object-orientation. Finally, the actual framework for transferrables will be presented.

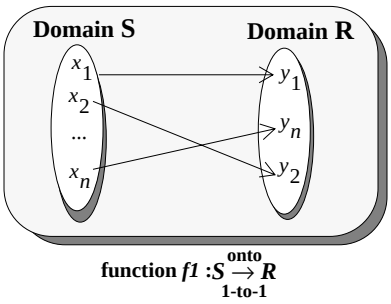
3.2.1.1 Domain Compatibility and Incompatibility

There are two major problems to be solved: byte order and lossless domain mapping. The byte order issue can be resolved simply by defining a network byte order for all inter-machine traffic as specified by Abstract Syntax Notation (ASN.1) [4] and Remote Procedure Call (RPC) [3]. The lossless domain mapping problem, however, is more complicated and requires more extensive analysis. We turn briefly to a discussion of the theory underlying the problem and the desired results.

In the following equation, ***D*** is defined to be the set of all concrete data types over the networked application. ***S*** is defined to be the set of all distinct data types for machine S and ***R*** is the set for machine R.

$$\begin{aligned} &\text{Let} \quad \boldsymbol{D} = \{ d_1, d_2, \ldots, d_n \} \\ &\text{Let} \quad \boldsymbol{S} = \{ x \in D; (x \in D_x) \} \\ &\text{Let} \quad \boldsymbol{R} = \{ y \in D; (y \in D_y) \} \end{aligned} \tag{1}$$

By defining a "*bijectional*[1]" function *f1*, all data types from machine S map to one and only one compatible data type on machine R. This domain mapping is onto[2] and 1-to-1[3].



(2)

function *f1* : $\boldsymbol{S} \xrightarrow[\text{1-to-1}]{\text{onto}} \boldsymbol{R}$

This mapping only guarantees a lossless mapping in one direction, such that the mapping from domain **S** to **R** will result in no lost precision. This is not true in the reverse direction as stated by the following equation. Which shows that the mapping of *x* to *y* is lossless from machine S to R, because the physical size of datatype *x* is smaller than or equal to *y* (thus, no lost precision).

$$f1 \rightarrow M(S_x, R_y): \{ x_i \leq y_i \} \tag{3}$$

But, if the physical size of datatype *x* is smaller than *y*, the reverse mapping of *y* to *x* may possibly be lossy (possible lost precision).

To define a purely lossless mapping in both directions, the datatypes *x* and *y* must map to each other. The following equation modifies Eq. (3) to provide symmetry between the mappings. It specifies that for each datatype, there exists a symmetrical mapping between machines S and R.

$$\forall d_i \Rightarrow \exists d \, [M(S_i, R_i) \vee M(R_i, S_i)] \tag{4}$$

This shows that for each data type *d* in domain ***D***, there exists a proper mapping from machine S to R; and vice versa, R to S. The equivalence of all appropriate data types will result in closure on ***D***.

$$\forall D \, (d) \, ; \{ x_i \equiv y_i \} \tag{5}$$

Eq. (5) illustrates that for every datatype in **D**, the mappings between any two machines are equivalent (in either direction).

3.2.1.2 Supporting Concrete Data Types

Now, to acquire this closure on all data types, we will need an equivalent logical mapping for each data type on all heterogeneous machines being used in the computation. What we are saying is that the software **must** learn to think in concrete domains. Instead of using the typical built-in data types like `int, float`, etc.., the application must utilize absolute domains (e.g. `int16`, `uint16`, `int128`, `float32`, etc..). These absolute domains take on the same behavior as the built-in language data types (e.g. unsigned objects contain only positive values, similar conditions occur for overflow/underflow, etc..). The difference is that the applications are forced to specify the data types sizes, versus relying on the architectural representation. To accomplish this, each absolute data type is treated as an object. Each object has two levels of representations, one is the logical representation (as viewed by the application) and the other is the physical representation (as viewed by internal physical storage). Only the physical representation is stored. When using the object, the logical representation will be used[4].

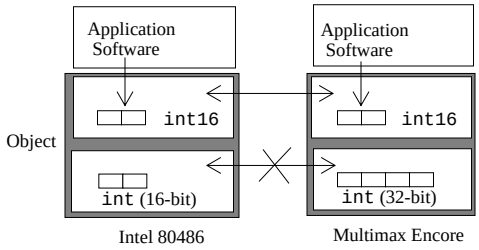


Figure 4 - Object Encapsulations: `int16`

As illustrated in figure 4, two inter-machine processes can communicate using the `int16` object. Since objects use encapsulation, the physical representation of each object is hidden. For both architectures, the physical representation happens to be each machine's **own** integer representation. In the case of the Encore, the logical representation can not exceed 16 bits. Now, if an application wanted a 32-bit integer on an Intel 80486 (`e.g. an int32` object), the

1. A function *F1: S->R* is **bijectional** if it is both one-to-one and onto. This is needed in order to define it's inverse.
2. If and only if each datatype in S maps to a data type in R, will the mapping be onto.
3. Each datatype in S maps to one and only one datatype in R.
4. The physical representation will always be greater or equal to the size of the logical representation.

physical representation would be a `long` integer. But, if using the `int64` object, it would have to accommodate for the additional storage required for that instance.

The concrete data types will be linked to the application during compiling[1]. These data types will not introduce run-time efficiency considerations. In the general case, the physical representation is typically a built-in data type. Cost factors can arise when a built-in data type is not available for a requested size (e.g. `int128` on an Intel 80486).

Now that we defined data types that have an absolute domain, the next question is *"How do you transfer them between machine* S *and* R?".

3.2.1.3 Transferrables

The transferable framework defines a protocol to encode and decode data structures in a language independent manner. In object-oriented languages and databases this term is known as persistence. We believe the support for persistent data structures is essential to develop serious distributed and parallel software applications, especially for non-numerical algorithms. The protocol supported by the transferable classes permits arbitrary data structures and scalars to be encoded for transfer between compatible and incompatible domains.

Each transferable will have the knowledge to encode and decode itself. A transferable is defined as a persistence descriptor followed by data. When encoding an object (e.g. `int16`), it will write the encoded data to a stream[2]. The receive end of the connection will read from the stream and decode the object. Each encoded object will consist of persistence descriptor followed by one or more bytes of actual data. The descriptor will indicate the concrete data type. It is important to note that for an `int16` object, exactly 16 bits of data (plus the descriptor) will be sent over the stream. This is true for 16, 32, 64, or 128 bit machines. This will allow the stream to be decoded properly at the receive end of the connection. The encoded data bytes will be in network byte-order as specified by the XDR[3] protocol [3]. For the case of an `int16`, three bytes of data will be transmitted over the network as shown with the encoded data in figure 6. This is an improvement over the ASN.1 transfer syntax [4]. Figure 5 shows that an `int16` object may be internally stored as a 32-bit integer on an i860. After encoding the object, the data is put in network byte order and is appended to the `int16` persistence type descriptor.

The major point is that even arbitrary data structures, with self-referential structures, can be moved with ease via the transferable classes. Without going into great detail here the fundamental observation is that all data structures have a spanning tree. A spanning tree can be constructed in time proportional to the number of vertices in the graph. Thus, it is possible to encode (linearize) an arbitrary structure in linear time and to decode (de-linearize) it in linear time. The OSI and RPC systems both require significant programmer intervention to manage the details of encoding and decoding data structures. This can make applications expensive and time consuming to develop. Another problem faced by both models is the inability to cope with different domains of architecture (as stated above in lossless domain mapping). The result is that the transferable objects support HC transparency by providing adequate scalar capabilities.

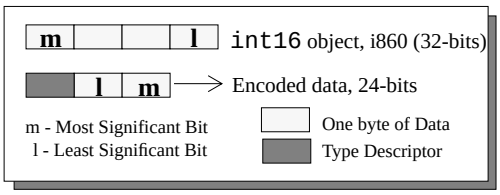


Figure 5 - Integer Representations.

To summarize, the transferable framework deals with the following issues:

- Encode arbitrary data structures efficiently and transparently with little or no user code required.
- Dynamically define data at run-time.
- Manage concrete types efficiently and cope with exceptions.

3.2.2 Network Communication Framework

Current research in networking protocols, such as DQRAP (Distributed Queuing Random Access Protocol), has shown that M/D/1 performance numbers can be achieved over a broadcast channel [21]. Such efficiency reinforces the idea of network connectivity utilization for any heterogeneous distributed computation. The idea of this framework is to provide a basic abstraction that provides network services through *generic requests*. In many respects, this is not completely new. The OSI model lays the foundation for one application interfacing with another application through a layered architecture. Each layer abstracting the details of the lower one through specific interfaces [7].

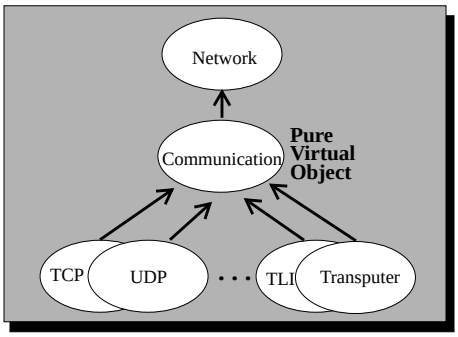


Figure 6 - Network Object Cluster

We are stating that a network connection should be treated as an object, where network services are handled through public methods. Using the ideas of the OSI model, this object is actually a cluster of objects. Each object being responsible for a certain aspect of the connection. Figure 6 illustrates a general cluster, allowing the application to interface the cluster through the `Network` object. The

1. Via the normal library linking process.
2. A stream is either a connection over a network or a contiguous array of memory that will eventually be sent over the network.
3. External Data Representation.

`Network` object is similar to the presentation/session layer of the OSI model. It uses an object called `Communication` as a collaborator. This object is an interface to the transport layer. The methods of this object, are for the most part, purely virtual. Therefore, through polymorphism, this object can be used to interface any transport layer.

A `Communication` object provides an interface to the transport protocol supported by the host. For many systems, this is merely a wrapper for the library of transport functions (e.g. Sockets and TLI). Providing a wrapper for a particular interface, is nothing more than deriving an object for it (e.g. an object that provides a TCP protocol by using sockets)[1]. However, many systems do not provide a transport layer, in which case a transport layer must be derived. For example, transputers have no transport layer. When a message needs to be sent, a channel is opened and the message is sent to it. This, however, results in performance degradation. Compute-bound processes that are ready to use the CPU are blocked until the long-winded communication is ended. A derived transport layer object that supports packet fragmentation and virtual connections would allow the communication cost to be amortized over time and allow other Processes to use the CPU in the process.

When establishing a network connection, we may not be physically connected to the destination machine. For this case, the assistance of a helper class (known as a collaborator), could be used (`Routing` object). This object can be used to abstract the routing policies of the application. We have found it useful to include this object in to the `Network` object's cluster, using it as a collaborator. This eases the burden on the application, it just passes a logical name to the `Network` object. The routing to the appropriate machine, via intermediate nodes, will be handled in the encapsulation of the `Network` object (via the routing collaboration).

The advantage of this approach is that the network object is easy to integrate into any application, along with being easy to port to new platforms through inheritance.

3.2.3 Locking Framework

Although there have been attempts to standardize locking mechanisms (e.g., POSIX), there will undoubtedly always be systems that expand on the capabilities provided by the standardized mechanism. Similarly, when the same types of mechanisms are provided, they tend to vary across platforms. Examples of mechanisms are semaphores, spin locks, and barriers. In contrasting the same two systems that were compared earlier, the Encore Multimax and Sparc 4 (see "Heterogeneous Computing Issues" on page 1), both system support semaphores; though their system interfaces are different. However, spin locks and barriers are only supported by the Multimax. To expand on the differences, the Multimax supports both process spin locking and process blocking. The fact is that there is no consistency between heterogeneous machines. This makes software reuse and portability difficult.

The locking framework provides a machine independent object that is used by the application to perform locking mechanisms via generic requests. The object will use the most efficient means for the machine that it exists on. Here again polymorphism will be used, where the locking object uses virtual member functions to execute the proper machine's locking code (see figure 7). If certain generic interfaces need to be provided to the application (e.g. barriers) and a platform does not support them, then the platform specific code must provide that functionality from existing interfaces (e.g. integrating counting semaphores with a wrapper to create a barrier).

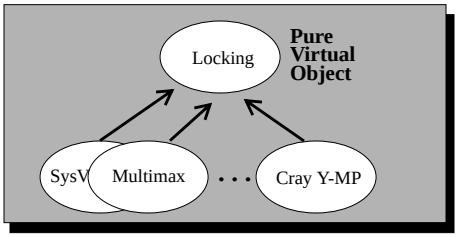


Figure 7 - Locking Object Cluster

The advantage of this framework is that it maximizes flexibility through framework reuse across machines and provides portability to the application code (as far as locking mechanisms are concerned). Adding a new machine to the network is as much as reviewing the current derived objects, if none of them meet the machine's interface, derive just the platform specific code into a new object.

3.2.4 Shared Memory Framework

Similarly to the locking mechanisms, shared memory interface provisions for one machine are rarely consistent with another heterogeneous machine (if provided at all). The main issue, is having a uniform interface that allows information sharing between one or more processes on the same machine. This will provide a transparent and reusable platform for upper lying software. The ideal medium is the host machine's shared memory, though if not present, message queues can be used. The Sparc provides a System V shared memory interface, while the Multimax provides an entirely different interface[2].

Both interfaces are different in nature. For example, the System V interface requires a shared memory segment to be created in the kernel. Once created, any process that knows the shared memory identification number, can attach to or de-attach from it. The segment must be explicitly removed. If not, it will remain in the kernel indefinitely. However, with the Multimax, a process creates the shared memory segment allowing only the processes that know the segment address to access it. If the segment is not removed explicitly by the process, it will be released when the process terminates. In order to provide an abstraction for the shared memory interface and usage differences, which are radically different between implementations, this framework must provide a

1. This is accomplished through inheritance. Polymorphism is used to access the derived class transparently.

2. The Multimax does support the System V shared memory interface, but is not as efficient as it's own style shared memory interface.

uniform interface (see figure 8).

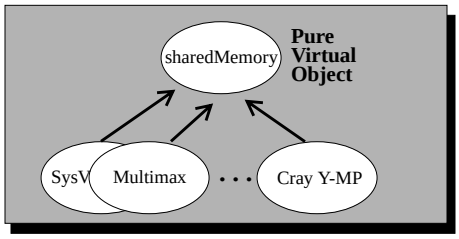


Figure 8 - SharedMemory Object Cluster

This framework increases shared memory code reuse between heterogeneous machines along with providing a more portable interface to the application code.

4 Heterogeneous Environments

High-performance heterogeneous distributed and parallel computing is emerging as a new paradigm [10]. For this type of research to have wide spread commercial use, there are certain aspects that it *must* have: portability, integrability, extensibility, interoperability, and reusability. These must be integrated in to simple systems engineering techniques to be useful in today's changing market place.

Each of the frameworks use a strict interface definition that relies on generic requests. The model stresses the importance of creating objects not to meet mythical future needs, but only under the present demand where the requirements are known. This ensures that a design contains only as much information as needed and avoids excessive complexity. Since the design is concrete and the logical relationship between objects is explicit, it is easier to understand, evaluate, and modify the design. Extending the design is typically deriving one or more objects.

The major objectives of increased software extensibility and reusability must control the complexity without effecting performance adversely. Many examples of the OO trend exist today, most commonly in operating system development (e.g. Microsoft's Cairo and IBM's Taligent object-oriented operating systems).

The dynamic binding and inheritance of object-oriented technology can be exploited to increase software reuse and hide architectural/protocol specifics [5]. Thus, providing a solid layering effect to provide higher transparency. To illustrate this, figure 9 shows an application with seven processes executing over three different platforms. The `Communication` object provides the same interface to all processes, eventhough different transport layers are being used. In addition, the `SharedMemory` object provides transparency, eventhough the underlying specific code is different[1].

Though we contest that systems can be built well with procedural programming languages, these approaches tend to lead to novice designs that lack modularity and are littered with regressions to global thinking (e.g. gratuitous global variables, unnecessary pointers, and an inappropriate reliance to other code modules). This leads to great difficulty in achieving a high level of software reuse over heterogeneous machines.

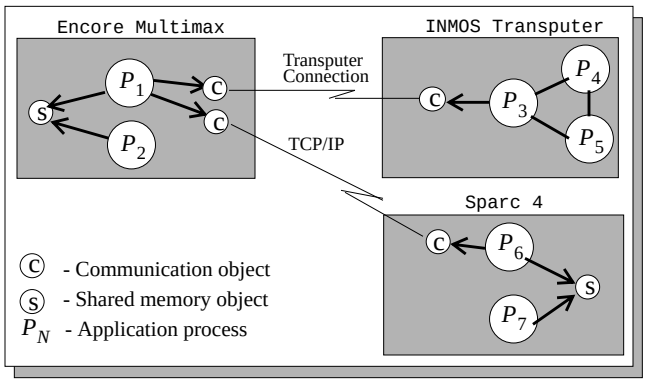


Figure 9 - Platform Specific Run-time Binding

Often these abstractions are not apparent because of our focus on designing algorithms to perform the desired computations rather than designing levels of abstractions. In addition, procedural abstractions are too small and not generalized in nature. The extent is that software reuse suffers effecting user productivity [17]. This results in increased development times when porting to newly enhanced hardware platforms and adding functionality to base code over all platforms.

5 Relation to Other Research

Both parallel processing and heterogeneous distributed computing has been an area of research for some time [12][23]. Recently there have been attempts in combining the two [6], known as *metacomputing*. When surveying the state of the art, several basic categories emerge. Each of the systems are not mutually exclusive; some fall into more than one category. The categories are: Remote execution [3][20][23][25], heterogeneity [6][12][18][19][20][22], and portable parallel processing [6][12][13][22][24].

Several programming models and distributed environments have been developed in the past for HC [6][12][13][22]. Each of these systems provide a distributed and parallel environment to the application code.

Excluding Mentat [13][22], each of the systems are focusing on procedural implementations (of the system itself). The general techniques and methods of implementing a procedural system are difficult when dealing with efficiency and understandability in HC environments. Another issue with all the current systems is the lack of consistent domain representations across heterogeneous machines.

Recently the OMG[2] has been defining the CORBA[3] standard. This standard supports the distribution of objects

1. Note that distributed-memory MIMD machines do not support shared memory. The shared memory framework must provide a virtual shared memory segment using message-passing.

2. Object Management Group (an alliance of companies).
3. Common Object Request Broker Architecture.

over heterogeneous networks. Through ORBs[1] located on each machine, the standard allows object methods to be invoked from anywhere on the network. Eventhough our model focuses on heterogeneous parallel processing, the communication framework can easily be used to provide the ORB to object connections. In addition, the transferables can be used to provide lossless datatype domain mappings for applications.

Our approach relies on portability, reusability, and extensibility while providing a high level of interoperability. What distinguishes our work is that metacomputing is simplified using the four *basic* frameworks. Not only does our model relieve the application of complex coding issues, but dynamic data migration, with lossless data domain mappings, is transparent to the application during run-time execution. This eliminates data type precision side-effects when using heterogeneity.

6 Conclusions

HC provides a wide range of new opportunities for distributed and parallel computing applications [15]. We must provide a common framework that provides for a set of fundamental components that are well defined, easy to integrate, and efficient for parallel processing in heterogeneous environments.

The generic modeling framework offers maximum software reuse and platform transparency. New platforms are easily integrated in to HC environments by deriving the appropriate super classes, allowing run-time binding, and providing total transparency to upper layer software. This increases reusability and time to market. The result is that the model allows a wide range of diverse high-performance machines to work together using a system built of basic foundations, general techniques, and clear methods.

7 Acknowledgments

We would like to thank Mohinder Chabbra and Jeffery Steele of AT&T Bell Labs and George Kutty of UCSB for their countless hours of proof reading and useful suggestions.

8 References

[1] W. O'Connell, G. Thiruvathukal, T. Christopher, "Distributed Memo: A heterogeneous Parallel and Distributed Software Programming Environment", Tech. Rep. TR-HDPP-1, Dept. of CS., IIT, Dec., 1993.

[2] S.R. Ahuja, N.J. Carriero, D. Gelernter, V. Krishnaswamy. "*The Linda Machine: Concurrent Computation*", Chapter 36. Plenum Press. 1988

[3] J. Bloomer, "*Power Programming with RPC*", O'Reilly and Associates. 1991.

[4] E. Blossom, "Decoding ASN.1 Transfer Syntax", *The C Users journal*, Sept. 1991, vol. 9, num. 9, pp 57-63

[5] G. Booch, "*Object-Oriented Design with Applications*", Benjamin-Cummings, 1990.

[6] V. Sunderam, "PVM: A Framework for Parallel Distributed Computing", *Concurrency: Practice and experience*, vol. 2, no. 4, Dec. 1990, pp. 315-339.

[7] A.E. Conway, "Performance Modeling of Multilayered OSI Communication Architectures", *Proc. of the IEEE Int'l Conf. on Commun.*, Boston, MA, vol 2, sess. 21, paper 1, pp. 651-657, June 1989

[8] N.J. Carriero, D. Gelernter, J. Leichter "Distributed data structures in Linda", *Proc. ACM Symp. Princ. of Prog. Languages*, January 1986.

[9] C.J. Date, "*Introduction to Database Systems*", vol. 1, ed. 4, pp. 371

[10] A. Deogirikar, Keynote Speaker, *TOOLS USA '93* Conference on Object-Oriented Technology. Santa Barbara, CA. Summer 1993.

[11] R.F. Freund, H.J. Siegel, "Heterogeneous Processing", *IEEE Computer*, June 1993, vol. 26, no. 6, pp. 13-17

[12] D. Gelernter "Generative Communication in Linda", *ACM Transactions on Parallel Languages and Systems*, Vol. 7, No 1, Jan. 1985, Pages 80-112.

[13] A. Grimshaw "Easy-to-Use Object-Oriented Parallel Processing with Mentat", *IEEE Computer*, May 1993

[14] A. Grimshaw, W. Strayer, P. Narayan "Dynamic, Object-Oriented Parallel Processing", *IEEE Parallel and Distributed Tech.*, Sys. & Apps., May 1993

[15] A. Khokhar, et al., "Heterogeneous Computing: Challenges and Opportunities", *IEEE Computer*, June 1993, vol. 26, no. 6, pp. 18-27

[16] P.S. Kritzinger, "A Performance Model of the OSI Communication architecture", *IEEE Trans. on Commun.*, vol. COM-34, no. 6, pp. 554-563, Jun. 1986

[17] J.A. Lewis, et al, "On the Relationship between the Object-Oriented Paradigm and Software Reuse: An Empirical Investigation", *Journal of OO programming*, july/aug 1992, vol. 5, no. 4, pp. 35-41.

[18] A. Ghafoor, J. Yang, "A Distributed Heterogeneous Supercomputing Management System", *IEEE Computer*, June 1993, vol. 26, no. 6, pp. 78-86

[19] J.R. Nicol, et al., "Object Orientation in Heterogeneous Distributed Computing Systems", *IEEE Computer*, June 1993, vol. 26, no. 6, pp. 57-67

[20] D. Presotto, R. Pike, K. Thompson, H. Trickey, "Plan 9, A Distributed System", AT&T Bell Laboratories, Murray Kill, NJ., *Computing Resources Archives*.

[21] W. Xu, G. Campbell, "A Distributed Queueing Random Access Protocol for a Broadcast Channel." *Comp. Comm. Review*, ACM SIGCOMM, Oct., 1993.

[22] A. Grimshaw, "Meta-Systems: An Approach Combining Parallel Processing and Heterogeneous Distributed Computing Systems", *Proceedings of Int'l Parallel Processing Symp., Workshop on Heterogeneous Parallel Processing*, 1992.

[23] D. Notkin, et al., "Heterogeneous Computing Environments: Report on the ACM SIGOPS Workshop on Accommodating Heterogeneity", *Comm. of the ACM*, vol. 30, no. 2, pp. 132(8), Feb. '87.

[24] J. Boyle, et al., *Portable Programs for Parallel Processors*, Holt, Rinehart and Winston, NY, 1987.

[25] B.N. Bershad, et al., "A Remote Computation in a Heterogeneous Environment", Tech. Rep. 87-06-04, Dept. of CS, Univ. of washington, Seattle, June, 1987.

1. Object Request Brokers.