# Easily Programmable Shared Objects For Peer-To-Peer Distributed Applications

**John Huebner**
School of Computer Science
Carnegie Mellon University
Pittsburgh, PA 15213 USA
jh6p@cs.cmu.edu
http://www.cs.cmu.edu/~jh6p

**Brad A. Myers**
Human Computer Interaction Institute
Carnegie Mellon University
Pittsburgh, PA 15213 USA
+1 412-268-5150
bam+@cs.cmu.edu
http://www.cs.cmu.edu/~bam

## ABSTRACT
This paper presents our experiences in implementing PERSON, a toolkit for adapting single user applications into multi-machine multi-user applications. This is achieved by providing a way to share objects in a peer-to-peer model using a programming model that emphasizes values rather than functions and ties the values together with constraints. This encourages a modular and declarative style of program design.

### Keywords
Toolkits, Distributed Applications, Amulet, CSCW, Constraints

## INTRODUCTION
Programming a multi-user, distributed application is significantly harder than writing an application that executes on a single machine for a single user [11]. If not using a groupware toolkit, the programmer has to explicitly create, open, and close sockets. Once the sockets are open, the programmer often has to deal with converting local data structures into byte streams to be sent over the network. This is called *marshalling* the data. On the receiving end, the programmer will have to unmarshall data.

PERSON, which stands for PeeR-to-Peer Shared Object Network toolkit, is able to convert constraint-based single-user applications to multi-user applications with only a few lines of code. It uses a peer to peer networking model to simplify application design. Users can join and leave at any time in any order without waiting for a moderator. There is no master server. The freedom from central authority is useful for a spontaneous gathering, however, PERSON also supports a client/server relationships by allowing one host to become the central hub through which all messages pass. When a participant drops out, the system cleans up automatically. Sharing is achieved by distributing copies of the important objects in the application, one per machine, and using cross-machine equality constraints to keep the important values of the objects the same on all machines.

PERSON uses an optimistic consistency strategy, of conflict detection rather than conflict avoidance, to allow reads and writes to be executed immediately. Conflicts detected using Lamport's logical clocks[6] to detect stale data. A deterministic algorithm is used to determine the final state after stale data is received.

The process of contacting a new participant, and getting all of the current versions of the objects to them is handled transparently by PERSON. The object name space is shared across a group of hosts. The programmer need not manage multiple copies of objects, or send updates since this is handled by PERSON.

### Amulet
PERSON is written in the Amulet user interface toolkit [8] in C++. Amulet has structured graphics, which is a drawing model that requires every image on the screen have an object associated with it. This allows the toolkit to take over many routine maintenance tasks such as refreshing the screen. Amulet objects have lists of value pairs called slots. A slot has a name and can hold any type of value. Amulet is designed so that programmer's can specify the relationships between values using arbitrary constraints. Most of a program in Amulet is defined by the values of the object slots. With this approach, there is little need for callback methods.

To the Amulet toolkit, we added a shared object system that allows the programmer to network an application simply by specifying which objects and which slots in those objects need to be shared. For example, one way to get a start button to become invisible when the game starts would be to write a constraint that makes the start button invisible when the ball becomes visible. Instead of needing a remote procedure call to tell the remote player to begin, the programmer can just make the ball visible. PERSON will propagate this change to the remote players, where the constraints on those machines will cause their start buttons to disappear.

Since we already had a large body of programs written in Amulet, we were able to convert some of them to be distributed, and observe the difficulties. The largest difficulty resulted when the variables that needed to be shared were in global variables of built in types rather than in the slots of objects. Conversion was notably easier when the original application had been programmed to have Save and Open commands, because the list of slots to save could be reused.

### Peer-to-Peer

There are two major design approaches for distributed applications: peer-to-peer and client/server. In the peer-to-peer model, all communicating hosts share equal responsibility for maintaining network services. In the client/server model, central servers provide services to the client. Client/sever is the dominant model for most toolkits today. There are social consequences to any network architecture. One consequence of client/server is that the central authority, in charge of the server, has a great deal of control over how, and when, the system is used.

Although the benefits of client/server are widely known, the maintenance of a central server can sometimes be a burden. The end user may not have access to a machine with high availability and an unchanging address. If such a machine is available, the users may need to request permission to run their software on it. In addition, client/server lends itself to designs where the server user must make decisions that strongly affect how the client users may use the system. This is useful for hierarchical organizations, but inappropriate for a group of peers collaborating or playing together.

In the strict client/server design, objects are stored separately from the host that uses them. For example, CORBA stores objects in one location and does remote procedure calls on the methods [9].

A peer-to-peer model enables designs where each user is responsible for their local resources and views. For many programming tasks, such a cooperative authoring, it makes more sense for the authors to manage communications themselves with all members as peers. Peer-to-peer is also a natural model for designing games, where, each player is often equal to the others.

In PERSON, responsibility for shared state is shared equally among the peers. Once an object is shared, a copy of all shared objects on each local machine which is viewing the object. All machines are equally responsible for storing or updating that object. In contrast, many peer-to-peer systems, such as Microsoft or Apple local file sharing, are constructed as distributed client-server systems, in which each peer accesses remote resources as a client and serves its own resources for others.

Although shared objects are stored on each machine, PERSON still allows the programmer to connect the hosts in whatever way he sees fit. By programming one host to always wait for the other hosts to connect first, and requiring the other hosts to connect to a known address, the programmer could create a client/server network topology. However, the objects would still be shared between the server and the client, rather than the server having full control over the objects. Since there is no need for a central server, the programmer can write one application instead of two, which simplifies the problem of maintaining consistent state on each application.

In order to allow existing Amulet programs to be quickly and easily converted to multi-user distributed applications, we wanted to ensure the freedom of a programmer to write a program without any concern for how information is sent. To meet this goal, PERSON hides both send and receive messages, as well as the connection process. PERSON also supports manipulating arbitrary types of data so that values can be sent across networks and among different types of platforms (Windows, Macintosh, and Unix). The programmer only needs to call one procedure for each peer added and one for each object shared. The rest is managed by the system. A programmer who is concerned about performance can concentrate on minimizing the amount of shared data, rather than designing network protocols.

The next section discusses related work. The section after that highlights the important features of our system. This is followed by an example illustrating how a single machine pong game was converted into a distributed application running on multiple machines. The sections following that will review the details of the implementation, beginning at the lowest level and continuing to the type of applications that can be written with our toolkit.

### Constraints

Amulet programs are declarative in nature and rely heavily on constraints to maintain the proper relationships between values. This declarative nature hides the value propagation mechanism, which allows PERSON to include network propagation transparently. For example, The ball in the pong game is constrained to be invisible if and only if the start button is visible. The start button is networked. Thus when one user presses the start button, it becomes invisible, and the local ball becomes visible. Because the start button is networked, the remote start button also becomes invisible and triggers the same constraint on the remote machine.

Even relatively complex cases are handled adequately. For example, consider the case with the start button and ball above. As shown in figure 2, when the start button on machine A (labeled Button A) is made invisible, then the update will be sent to the start button on machine B (labeled Button B) and the constraint will update the ball on A (labeled Ball A).

If the network update of the Ball B arrives before the network update of Button B, then the constraint will simply re-set the same value, and propagation will stop.

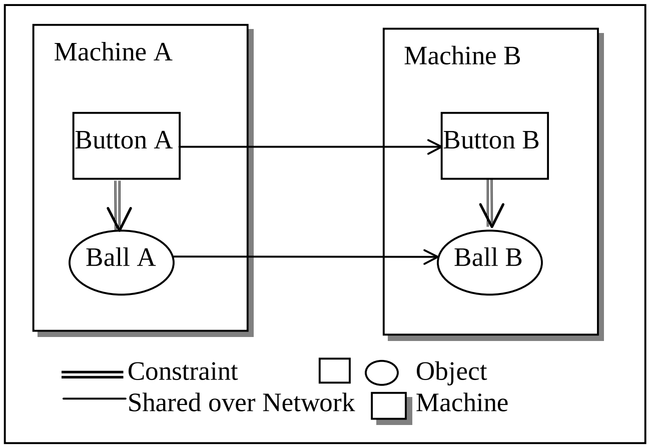


**Figure 1: Overlapping Constraints and Links**

If the update to Button B triggers the constraint before the network update to Ball B arrives, then the constraint will trigger an update from Ball B to Ball A. This will result in both Ball A and Ball B being updated by the network. Since the values will be the same, no constraints will be triggered, and the propagation will stop because it is prohibited from traveling back across the share it came in on.

### Topology

The network topology is an arbitrary, undirected, graph, where each machine is only connected to the other machines that it names explicitly in a call to the `Add` method.

A star topology could be created if users agreed to all connect to one central machine that waited with `Wait` calls. This would approximate client/server.

The more interesting topology that we allow is chains. In a chain, each machine connects to the next. Even loops are permitted.

Under the current implementation, leaves of the graph may come and go, but nodes that connect two or more other nodes could cause a partition in the network if they dropped out. Each node only relays changes to the immediately adjacent nodes.

### Conflict resolution

The current system has simple equality constraints that guarantee that the value chosen by all hosts will be the same, and will be a value sent by one of the hosts. Although this does not prevent one host from overriding another's changes, it does prevent corrupted data.

Inconsistencies may not matter and people may mediate their own actions, as noted in early GroupKit work, [5]. That paper also notes that if visible feedback of conflicts is presented, then people are able to resolve the conflict. Our policy ensures that the result converges to some host's input. If the local machine did not win, the user will perceive the presence of the other users and act to coordinate with (or over-rule) them.

## RELATED WORK

One issue with multi-platform distributed applications is the marshaling and unmarshalling of data to support varying data storage designs such as byte ordering. GroupKit avoids marshalling data into byte streams by requiring the programmer to use Tcl/Tk, a language where all types are represented as character [10]. VisualOblique [2] and Programmer's Playground [4] both provide abstractions that transparently transport data from one host to another, but both require that the application be written in from scratch in a new language. In contrast, we had written applications in amulet before the addition of network capacity and concentrated on converting these programs rather than writing new ones.

Another important issue in distributed Applications is the design of the naming services to allow applications to find the remote resources they need. For systems that share values rather than remote procedures, this takes the form of the data name space used in expressions and assignments. VisualOblique uses top level windows, which it calls forms, as collections of widgets as well as the basic unit of distribution. Values are stored as properties of widgets in form instances. The server keeps an array of handles to instances of forms, forming a hierarchical name space.

Programmer's Playground allows the programmer to package code into *modules*, which are components with public data that can be accessed by other modules, called *published interfaces.* Programmer's Playground allows programmers to connect modules using a graphical user interface. Two modules are connected by sharing elements from the public interfaces of each module. The programmer can specify callback functions for each public data structure. A connection manager module enforces access privileges.

PERSON allows grouping objects into sets, called network groups, which are only visible to hosts who are members of that group. Within network groups, our namespace

One of the most difficult challenges of distributed shared data systems is the maintenance of consistency with respect to cause and effect when changes affect distant data. Bharat & Hudson were able to achieve a concurrency in all but read operations in their Doppler distributed constraint system, without sacrificing any causal consistency. To achieve this, they built an elaborate infrastructure into Doppler that tracked state changes with vector clocks, in a data structure that kept copies of previous states for reference, [3]. However, Doppler still required locking data during reads, which requires messages to make a round trip between hosts before a read can proceed.

There are also attempts to create high-level programming abstractions for client/server systems. Tools such as Java's Remote Method Invocation [12] and CORBA [9] support abstractions that hide the byte stream behind local stub (client side) or skeleton (server side) proxy objects that convert to and from local data structures. These frameworks introduce as much complexity as well as hiding it, however. The setup for these frameworks often involves two or more objects and several method calls on the client side, and the server can be as complex. The main improvement is that these tools can be used for a variety of objects. Furthermore, the programmer still has to invent a protocol of method invocations and or messages to pass back and forth.

Microsoft's DirectX gaming toolkit, uses a Lobby server for establishing connections. Even for peer-to-peer messaging [7] requires a lobby server to establish a connection. GroupKit uses a Registrar to connect its clients [11].

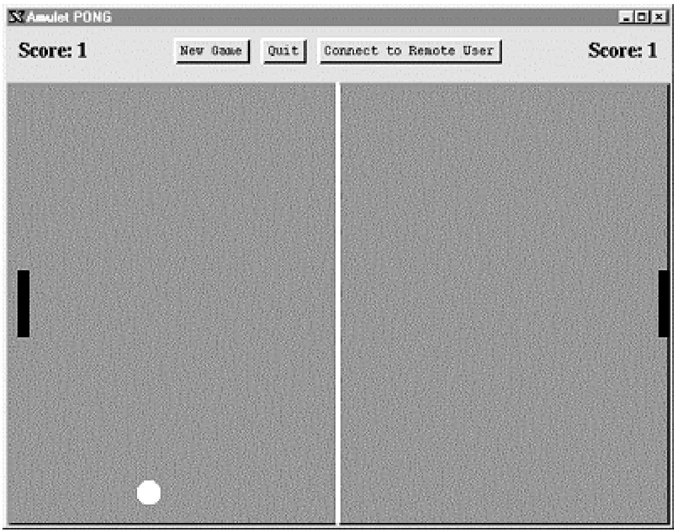


**Figure 2: Pong-A sample networked application**

## EXAMPLE

In a sample program, a Pong Game was converted from single machine to networked by adding a few lines of code. First the programmer specifies how the objects are to be shared and then shares the objects. Other hosts may be added before or after objects have been shared as shown below:

```
Am_Network.Share(paddle1,"pad1");
Am_Network.Add(hostname)
Am_Network.Share(paddle2,"pad2");
Am_Network.Share(ball,"netball");
```

The hostname is just a string which can be entered by the user as either a machine name (e.g.: www.cs.cmu.edu) or its dotted decimal form, (e.g.: 128.2.209.79). The Add method can appear anywhere in a program. The human readable Internet address format has become widely used with the spread of the World Wide Web which uses it as part of the Universal Resource Locators. Even Microsoft has added IP addressing as an optional part of its file-sharing IDs.

When paddle1 is shared, it is registered with the low level as an object to share across machines.

When the value of a slot in an object changes, the new values of the object's slots are sent to the other machines that are sharing that object. Whole objects are not sent during these updates. Amulet has a predefined slot called Am_SLOTS_TO_SAVE, which is a list that can be added to any object to indicate the slots on that object that should be stored when the object is saved to a file. The list usually includes properties that change from object to object such as color or position information. There are many slots that are omitted such as the object's the line thickness, or border color, which are never changed within that particular application and can get their values from the parent object. PERSON uses the Am_SLOTS_TO_SAVE list to determine which parts of an object's state need to be saved to disk. Thus, if the program was written with a "Save" command, there is no need to specify Am_SLOTS_TO_SAVE. The network system will send the slots already specified in Am_SLOTS_TO_SAVE. However, since Pong does not normally save games, these lists had to be added. The following code is added:

```
ball.Add(Am_SLOTS_TO_SAVE, Am_Value_List()
             .Add(X_VELOCITY)
             .Add(Y_VELOCITY)
             .Add(Am_VISIBLE)
        )
```

## IMPLEMENTATION

The following sections detail the design used by PERSON and the lessons learned. First the underlying Connection layer that supports the network abstractions in the toolkit is described. This is followed by a description of how different values are sent across the network. The next section explains how lists and other composite types can be easily constructed using high-level functions. The Object section presents the sending and maintenance of objects. The section after that explains how the objects are kept consistent using demons which trigger in response to changes in values. Finally, here is a brief discussion about the network topology and our conflict resolution strategy.

### Connection Layer

The `Am_Network.Add` method used in the example, relies on an underlying connection service which we also wrote. The connection service handles the establishment and maintenance of the connections between all the participating machines. A socket is opened for each connection and maintained as long as the machines need to communicate. Since each connection between two hosts will have a separate socket, there is no cross-talk and data can come in from multiple hosts simultaneously without difficulty.

To establish a connection with a remote machine, `Am_Network.Add` calls the static method `Am_Connection::Open`, which returns a connection. There are two ways to call `Open`. If it is called with a hostname string, it will try to establish a connection with the application on the machine with that hostname, and fall back to listening for incoming connections from any host if that fails. If it is called without any parameters, it will immediately begin to listen for any incoming connection.

This has the convenient result that if both sides call `Open` with the other's address, a connection will be established regardless of who calls `Open` first.

The passive form of `Open` can be used to create a server like behavior where the application will wait for unknown incoming connections.

### Sending Values

PERSON supports a strong but flexible type system, where all variables can be stored in a generic union type: `Am_Value`, which has an attribute that can be used to determine the type of data that it contains. This allows us to have an `Am_Value_List` type that can store different types of data in each position of the list, and allows us to have object slots that can store any data type.

Before a new data type may be sent or received, its type is registered. We register all of the built in types such as char, bool, short, int, long, float, double, and string as well as the composite types Am_Value_List and `Am_Object`. In most cases this will be all that the programmer will need, because new values of any type can be added to an object or list dynamically. If the programmer wishes to send a new type, he may write a marshaller and register it. To write a new marshaller for a composite type, the programmer must use the connection's Send and Receive methods to send the primitive types that make up the composite type. This is how the Am_Value_List and Am_Object marshallers are written. For example, the programmer could use this technique to implement Complex Numbers by sending a pair of floats. A marshaller for the Complex Number type would extract the real and imaginary components from the custom data structure and send them, in order, by using:

```
my_connection.Send(my_real_float);
my_connection.Send(my_imag_float);
```

The unmarshaller would then receive the basic types using

```
my_real_float=my_connection.Receive();
my_imag_float=my_connection.Receive();
```

A type is registered with a single method, which has three parameters: the type code that is identifies the type, and the functions to marshal and unmarshall that type. These are stored in association lists. Since the two functions are registered together in the same call as the type information, every registered type is assured both a marshaller and an unmarshaller that are consistent.

Only the marshallers for built-in types use the sockets layer. The `Am_Connection::Handle_Input` method, called from the main loop, checks sockets for incoming data and calls the marshallers. This is the only method, besides marshallers, in `Am_Connection` that uses socket calls to perform its duties. Marshallers for composite types use the `Send` and `Receive` services.

Once a connection is established, the marshaller is able to send any value supported by the system via the same `Send` method, and receive any supported value using `Receive` method. The `Send` method sends the type code across the network, then calls the appropriate function pointer for sending that type.

At the receiving side, when a new value is received, the type is used to find the appropriate the unmarshaller function, using another association list. This function is called to build the correct value from the network stream.

Once a new variable is received it is put onto a queue for the connection that it came in on and the callback function that is registered for that connection is called to process the variable

The callback function may use `Receive` to get the value off the queue. The default marshallers for integers use the BSD network order functions. We wrote similar functions for float and double, which assume IEEE format, since no network order functions were implemented for floating point types. Strings are sent as lengths followed by the character streams.

For example, if a program called `Send` with a four byte integer as a parameter, the Send method would detect that the type was 32 bit integer and send the type-code over the network to signal the arrival of a new integer. It would then call the marshaller registered for 32-bit integers. This marshaller would use the BSD function `htonl` to convert the integer to network byte order and then send the reordered four bytes over the socket. On the receiving end, the incoming data would trigger the handler for socket input, `Am_Connection::Handle_Input,` which would detect the type-code for a 4 byte integer, and call the unmarshaller for 4 byte integers. This unmarshaller would read in the integer and use the BSD function `ntohl` to convert the network byte order to host byte order. The newly received integer value would then be wrapped in an Am_Value union type and placed on the queue to be used by the callback function registered with that connection

Callback functions are responsible for popping the value off the connection's queue and acting upon it. The Am_Network layer's callback function does nothing but pop the object off of the queue, since the real action happens as a result of constraints that depend on the values of the object's attributes.

### Lists (an example of a composite type)

Types that could contain other types, such as Lists and objects, required special attention to handle recursion such as lists within lists, or objects within objects. We also want to prevent the system from blocking while additional data was on its way. In addition, both objects and lists needed to contain items of arbitrary type. To implement this we overrode the user-defined receiver callback function so that we could re-use the receiving system. The new list simply stores the handler for the parent list, along with the partially constructed parent list on a queue in the Connection. The unmarshaller constructs the child list and returns control to its parent's receiver callback function as its last action. The parent list handler then takes the newly generated child list from the queue, and inserts the child list into a slot in the parent list. Construction of the parent list continues where it left off. This supports arbitrary levels of recursion.

### Objects

The Am_SLOTS_TO_SAVE list contains an ordered series of slot keys. This list must be in the same order on each machine that shares the object. PERSON does not depend on the slot keys being the same, which is important, since users may request slot keys dynamically to add new slots. These dynamically allocated slots may be different on different machines.

### Application Level Semantics

PERSON uses *network groups* to determine which machines share a group of objects. A default group, `Am_Network`, is provided. A network group is a collection of machines that share a set of objects. When a program wants to start communicating with another program, it must add the internet address of the other program to one of its network group using the `Network_Group.Add` method.

The Add method takes a string argument as discussed in the example. The address can be entered in by the user, allowing these programs to be run even when there is no dedicated server, unlike the server required in GroupKit, VisualOblique, DirectPlay, and others. So long as the participants know the address of the machine of one other participant, they are able to join the group. There is no need to designate one user as the server user.

If the programmer wishes to wait for a connection without specifying a machine name, he may call Am_`Network.Wait`. This has the same affect as calling `Add` without an address, but is less confusing to read.

One potential use for network groups is in the division of labor as shown in figure #2. Network groups would allow a team of users on separate machines to divide a workspace

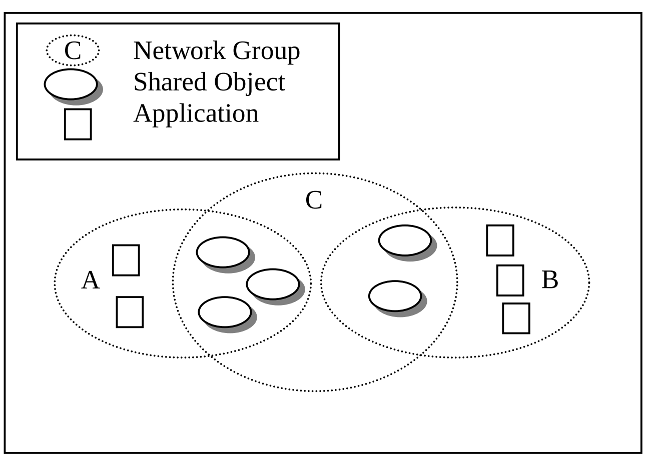


**Figure 3: Moderated Parallel Work.**

into parts, and have group A share the objects they are working on which are distinct from the objects that group B are working on. A small moderator group C could share all the objects.

It is necessary to `Share` an object to each group with which the programmer wishes to share the object. When the programmer calls `Share` on an object, it remains shared until the programmer calls `Unshare`.

The `Share` function is a method of the network group rather than the object so that objects need not be modified in any way. Share takes an object as a parameter. For example, the command to share the object `paddle1` is:

```
Am_Network.Share(paddle1,”pad1”);
```

The arbitrary string in the above example serves to uniquely identify the object across all participating peers. There is no required format, but it must be unique across all machines in the network group. When an object is shared, it is registered in an association list so that it can be referred to later. Both sides require the same registered object for the system to keep a shared object consistent. When an object is received, the slots specified in Am_SLOTS_TO_SAVE are overwritten with data from the network.

### Synchronize objects by using demons

When an object is shared, a reference to the object is stored and a demon is set on each of the slots listed in the Am_SLOTS_TO_SAVE list. If one of the slots in an object’s Am_SLOTS_TO_SAVE list contains another object, the network demon is set on the contained object as well and so on recursively.

When the value of a shared object’s slot changes, its demon is put into an execution queue. If multiple changes are made to the slots of that object, the demon waits in the queue until some other operation is performed such as requesting the value of one of the slots of the object, operating on a different object, or reaching the end of the event loop. At that time, the demon activates and sends the object to each of the participants in turn.

Before an object is received, it should be `Shared`. Once the object is shared on both sides, all subsequent references to that name will refer to the same object, regardless of which side sent the update. This way the system knows what to do with the new information. For example, it knows what constraints should be alerted of changes in slot values. If the programmer wants to be able to dynamically create objects on a remote screen, they program also needs to know that these new objects need to be part of the screen. In our design, this could be done by creating a group on both sides and adding the group to the desired window. Then objects could be added and removed from the group and appear on the screen as they would be expected to do.

When a shared object is received, a new slot called `Am_NET_IN_PROGRESS` is added which contains the address of the connection that sent the update. The network demons check for the presence of this slot and do not send updates to connections named in the Am_NET_IN_PROGRESS slot of the object being updated. This prevents loops involving an incoming update from triggering an update back to the host that sent the original update. This slot is deleted once the object is completely constructed and placed on the connection queue.

## FUTURE WORK

At the time of the writing of this paper, we have implemented the marshallers for all basic types as well as Strings, Am_Value_Lists and Objects. The establishment of connections and sending and receiving supported types are complete for the Am_Network_Group.

One way we hope to increase performance is by sending partial updates. Currently all slots in the Am_SLOTS_TO_SAVE list are sent every time the object is changed. In the future we may implement sending only the changed slots with their slot key and analyze the performance benefits.

In the future we plan to add more robust networking in the form of a new network group type that is fully connected, called `Am_Fully_Connected_Network_Group`. This new network group would ensure that all participants in a group see the same members in that group. The new group, which will be called, will demand that each node send updates directly to all the other nodes in its group. This design would require each machine to pass along the list of all members of that group so that each member could directly communicate with the others. This would provide fault tolerance for node failures and increased availability. For example, if host A connected to host B and machine B connected to machine C, machine B could drop out and leave machine A connected to machine C.

This is illustrated in figure 4, where Step 1 shows the initially unconnected nodes. Step 2 shows A connecting to B. Step 3 Shows B Connecting to C. In step 4, the current Am_`Network_Group` simply completes the connection between B and C, whereas `Am_Fully_Connected_Network_Group` would establish an additional connection between C and A. In step five, B looses its network connection. In step six, A and C are isolated under the current design, but still have a working connection in the fully connected design

We would also like to expand the types of objects that are supported to include automatic sharing of objects created from objects that are already shared. Other enhancements would include supporting the deletion of `shared` objects, either by simply unsharing them or providing some mechanism for safely deleting remote objects. We would like to be able to add slots to the Am_SLOTS_TO_SAVE list while an object is shared. We would also like to explore the implications of sharing visible windows directly rather than their parts as well as supporting updates to lists

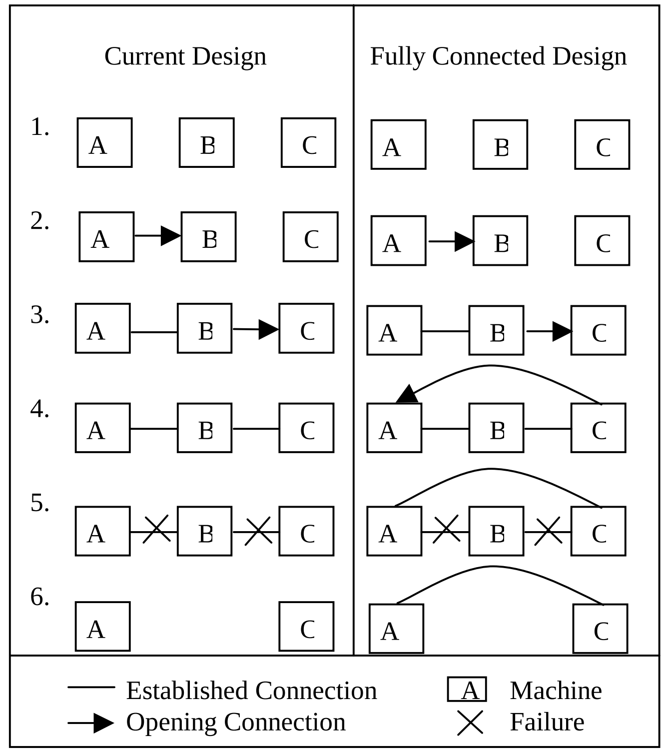


**Figure 4: Advantages of a Fully Connected Design**

where list members are added or deleted. Also applies to objects who have new sub-objects set in a shared slot.

.Other directions that we might take PERSON include sending command objects to support network undo as a concurrency control mechanism, as GINA [1] does, or making our naming scheme hierarchical.

## Conclusion

Our experiences support the evidence presented by VisualOblique and Programmer's Playground that the declarative programming style provides an excellent method for abstracting the complexities of designing distributed multi-user applications. In addition, we also demonstrated the usefulness of constraints in a distributed setting. Unlike VisualOblique which had a large number of callbacks, our system used constraints (Reference spaghetti code paper?). We simplified the naming process, while supporting information hiding through the use of network groups. Our session establishment process is basic, but capitalizes on the recent public familiarity with internet addresses because of the proliferation of the world wide web.

The largest problem we are experiencing is poor performance. We found that the choice of which slots to send was crucial to acceptable performance. In an early version of the pong game, we simply shared the position slots of the ball. This resulted in updates arriving a few seconds late, even on a our local LAN, which caused synchronization problems. When we shared the velocity of the ball rather than the position, the game became playable.

## ACKNOWLEDGMENTS

This research was partially sponsored by NCCOSC under Contract No. N66001-94-C-6037, Arpa Order No. B326. The views and conclusions contained in this document are those of the authors and should not be interpreted as representing the official policies, either expressed or implied, of the U.S. Government.

## REFERENCES

1. Berlage, T. and Genau, A., “A Framework for Shared Applications with a Replicated Architecture,” in *Proceedings of the ACM SIGGRAPH Symposium on User Interface Software and Technology*, 1993, pp. 249-257.

2. Bharat, K. and Brown, M.H. “Building Distributed, Multi-User Applications by Direct Manipulation,” in *Proceedings UIST’94: ACM SIGGRAPH Symposium on User Interface Software and Technology*. 1994. Marina del Rey, CA: pp. 71-81.

3. Bharat, K. and Hudson, S.E. “Supporting Distributed, Concurrent, One-Way Constraints in User Interface Applications” in *Proceedings UIST’95: ACM SIGGRAPH Symposium on User Interface Software and Technology*. 1995, pp. 121-132

4. Goldman, K.J., *et al.*, “The Programmer's Playground: I/O Abstraction for User Configurable Distributed Applications.” *IEEE Transactions on Software Engineering*, 1995. 21(9): pp. 735-746. http;//www.cs.wustl.edu/cs/playground/papers.html.

5. Greenberg, S. and Marwood, D., “Real Time Groupware as a Distributed System: Concurrency Control and its Effect on the Interface,” in *Proceedings of ACM CSCW’94 Conference on Computer-Supported Cooperative Work*, 1994, pp. 207-217.

6. Lamport, L. “Time, Clocks, and the Ordering of Events in a Distributed System.” *Communications of the ACM*, 1978. 21 (7): pp. 558-56

7. Microsoft, “DirectX Online Help.” 1997. www.microsoft.com.

8. Myers, B.A., *et al.*, “The Amulet Environment: New Models for Effective User Interface Software Development.” *IEEE Transactions on Software Engineering*, 1997. 23(6): pp. 347-365.

9. OMG, “CORBA 2.2 Specification.” 1998. OMG Technical Document formal/98-02-01

http://www.omg.org/corba/corbaiiop.htm.

10. Roseman, M. “Tcl/Tk as a basis for groupware,” in *Tcl/Tk Workshop*. 1993.

11. Roseman, M. and Greenberg, S., “Building Real Time Groupware with GroupKit, A Groupware Toolkit.” *ACM Transactions on Computer Human Interaction*, 1996. 3(1): pp. 66-106.

12. Sun, “Java RMI reference.” www.sun.com