SVMDCP Net Layers FAQ

December 8, 2003

Contents

1 What is SVMDCP?

SVMDCP (SVM Distributed CheckPointing) is a backend support for distributed checkpointing and rollback for SVM and SCC. If you don't know what SVMDCP is yet, please be directed to the SVMDCP homepage.

If you still don't know what SVM and SCC are about, please check out the SVM homepage and SCC homepage.

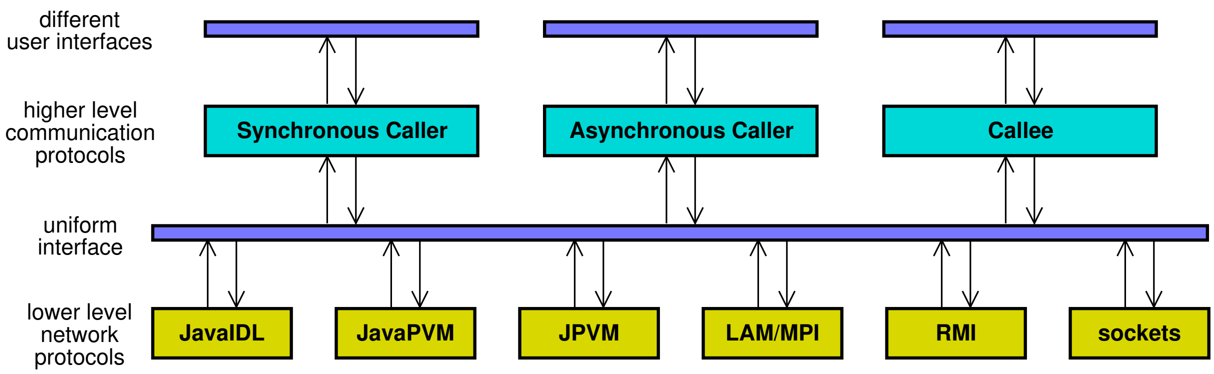

Figure 1: Layout of the net layers

2 What are the net layers?

The net layers include multiple underlying technologies of distributing computation, such as PVM, MPI and CORBA. Although some of them build themselves on top of others (e.g., CORBA is on top of sockets), they are regarded as peers in the net layers, and one can be substituted by another for specific reasons. There is no loss of functionality in the substitution.

The net layers also include protocols at a higher level: synchronous communication and asynchronous communication. These two protocols require support from those underlying technologies, and provide uniform interfaces to the programmers.

Between the two levels there is a well-established interface that hides the implementation details of the underlying technologies. It makes it possible that the choice of technologies at the lower level does not affect the applications built on the upper level. The programmers can plug-and-play with different network technologies without modifying their applications.

3 Why use the communication protocols instead of using the different technologies directly?

Network technologies have their pros and cons. It is very hard to tell which one is better than the others. This is the main reason for their co-existence for a rather long time. When planning a distributed system, the designers always face this problem of choice. For example, CORBA is more standardized and is widely accepted in the industry. However, it is not as efficient as other technologies, which are at a much lower level than CORBA. On the contrary, building a system directly on top of sockets yields high performance, but it is thus hard to communicate with other software because of a lack of well-established protocol. As another example, a network protocol fully implemented in Java is portable and makes the communication system architecture-independent. However, for a system dedicated to a specific type of hardware (e.g., an embedded system), the designers care more about performance, but do not mind using platform-dependent code (such as C code).

For a general-purpose network support, there is no definitive answer to this argument. A wise solution is to support all these technologies, and leave enough flexibility to the designers of specific systems.

4 As all these are supported, is the library hard to use?

No.

There are two kinds of users. The users at the upper level gain support for synchronous and asynchronous communication. The underlying network technologies are hidden from them. They create network objects at runtime, and use these objects as the handles to initialize their synchronous or asynchronous communication protocol. On the server side, they do need to choose a technology and create relevant objects of it. Then the server ID is published to the clients. With this ID, the clients automatically detect the protocol that the server uses, and communicate with it using a compatible network technology.

The users at the lower level use the network protocols directly to send and receive messages. The uniform interface flattens the learning curve and allows for applications that support multiple protocols.

For both kind of users, using the net layers is very easy.

5 Do you have a client-server example?

A server example is shown here:

```
// create an RMI network object
Network net = new RmiNetwork();
// initialize the network
net.initialize();
// get the object that represents this server
RemoteObject ro = net.getRemoteObject();
// print the server ID to screen
System.out.println("MyID = " + ro);

// suppose MyServer is a subclass of SemiblockingReceiver
// create a receive for incoming calls
SemiblockingReceiver receiver = new MyServer(net);
// starts the server, the current thread is blocked
receiver.start();

// finalizes the network
net.finalize();
```

The startup of the server is pretty simple. By creating different network objects (such as `CorbaNetwork`, `JavapvmNetwork`, `RmiNetwork` and `SocketNetwork`), the programmer easily implements the server on difference technologies. Of course this can also be done at runtime with dynamic class lookup.

The clients have to know the ID of the server before they issue calls to it. The server ID contains all necessary information. For the above example, the ID may look like rmi://localhost:1099/MessageReceiver.R1234208140. It denotes the protocol (RMI in this case), the host and port of the server, and any other specific parameters. The clients must use the same technology as the server. Fortunately, this is taken care of by an ID solver:

```
// convert the ID to a remote object
RemoteObject server = RemoteObject.fromString(ServerID);
// get a network object that is compatible with the server
Network net = server.getNetwork();
// initialize the client network
net.initialize();

// create a synchronous caller that uses this network
SynchronousSemiblockingBridge caller = new
    SynchronousSemiblockingBridge((SemiblockingProtocol) net);
```

```
// call the server with a string parameter
caller.call(receiver, "Hello World!");

// finalizes the network
net.finalize();
```

This piece of code on the client side does not depend on the specific technology that the server uses.

6 How many different network technologies are supported?

At the moment when this FAQ is written, the net layers include support for these:

- JavaIDL, a CORBA implementation distributed with Java 2 SDK 1.3 or higher.
- JavaPVM, now renamed to jPVM, a PVM (Parallel Virtual Machine) binding for Java.
- JPVM, a re-implementation of the PVM library using native Java methods.
- LAM/MPI, a partly MPI 2 compatible platform for high performance distributed computation.
- RMI, another CORBA implementation distributed with Java 2 SDK 1.3 or higher.
- sockets, native Java stream-based sockets.

Among them, JavaIDL, JPVM, RMIand sockets are 100% Java, while JavaPVM and LAM/MPI require support from corresponding C libraries.

7 What's the difference between blocking and nonblocking?

Every network technology focuses on outgoing messages and incoming messages. If when the sender sends a message to another host (a local process or a remote computer) it has to be blocked until the message reaches the receiver, we call this scheme *blocking send*. Of course, when the sender is blocked, it cannot proceed with its work (for a more clear definition, see below about blocking levels). On the contrary, if it does not wait until the receiver gets the message but continue with its work immediately, the sending is *nonblocking*.

It is similar for incoming messages. If the receiver is blocked until it actually gets a message, the scheme is *blocking receive*; otherwise, it is *nonblocking receive*.

Semiblocking is a mixture of blocking and nonblocking. Usually people want the sending to be nonblocking, as long as the messages eventually reaches the receiver intact and in their order. However, there is demand for blocking receiving. It is usually meaningful to wait for incoming requests or results without performing other task. In a word, semiblocking means nonblocking send and blocking receive.

For blocking schemes, it makes difference whether the *blocking level* is thread or process. If the blocking level is thread, like most network technologies such as CORBA and sockets, other threads in the same process are allowed to proceed with their computation. However, it must be noted that some libraries such as MPI is not thread-safe and does not allow their functions to be called by multiple threads. In this case, though the other threads are still active, they cannot participate in network communication during the blocking. If the blocking level is process, all the threads in the same process are blocked automatically until the sending or receiving is finished. JavaPVM is an example of this kind.

8 What's the difference between synchronous and asynchronous communication?

Synchronous calls and asynchronous calls are two different types of communications, where messages are sent and received on each node. For a *synchronous call* or *synchronous communication*, the originator of the call waits until the call returns. This usually means the receiver of the call performs certain computation and sends back the result. In case where no result is returned (or the result is *void*), the callee must still send back an acknowledgment, which informs the caller of the end of this call and allows it to proceed. The blocking level of synchronous calls is usually thread instead of process. While the caller thread is waiting for reply, other threads in the same process are active. They can even call network functions.

On the contrary, if the caller is not blocked and the call immediately returns regardless of whether or not the callee has finished its work, the call is *asynchronous*. Basically there are two schemes for the return of this call: polling and callback. If the *polling* scheme is supported, the caller can use certain functions to check if the result is available. It may periodically do this checking. Once the result is available, it may be retrieved from the buffer. If the *callback* scheme is supported, once the result is ready, the caller receives a signal from the system. This usually means a pre-specified method (*callback method*) is invoked. The polling scheme can be used to simulate the callback scheme (if it is not provided) with a little loss of efficiency, and vice versa.

9 How do the net layers transfer messages?

All the messages must be serializable. An outgoing message is serialized into a byte array and sent to the other side with whichever underlying technology. On receiving the message, the receiver must deserialize the byte array to obtain the original message. This also means the message must be copyable. If it contains non-copyable parts that require special treatment before serialization, its class must override the following methods in the Java `Serializable` interface:

- `private void writeObject(ObjectOutputStream out) throws IOException;`
- `private void readObject(ObjectInputStream in) throws IOException, ClassNotFoundException;`

A message may be sent with an integer *tag* value. The receiver may use the same tag value to retrieve it, or it may also use a *universal tag* to retrieve the first message in the buffer with any tag. As an example of the usage of tags, the synchronous caller uses the hash code of the current thread as the tag when it sends messages and receives results. (The callee sends back results with the original tag.) With this, the caller is thread-safe: a thread does not receive a wrong reply which is dedicated to another thread. For the protocols that do not support tagging, the net layers take care of simulating this support.

10 Why some C based technologies are much faster than Java native ones?

One can feel big difference in performance if he/she has used both C and Java. One main reason is that Java uses VM (Virtual Machine) to interpret byte code. Interpretation is of course much slower than direct execution of hardware instructions. Dynamic features of Java, such as class lookup, garbage collection and so on, are an other reason.

However, one who has tried to re-implement some Java functions with JDK itself should know that bad designs do exist in the Java library. For example, serializing a `Serializable` object in a standard way is about 10 time slower than doing serialization by hand. This is why the net layers actually provide two types of serialization:

- If the object to be serialized is a well-known type, such as `String`, `Integer` and `byte[]`, the serialization result will be an array with a leading byte denoting the type and the contents following this byte.

- Otherwise, the first byte will be -1 and the standard serialization result will be following it.

Unfortunately, this is not the only example that I have found during the implementation of the net layers.

As a result, the C technologies are much faster. In particular, the JavaPVM network layer is about 15 - 20 times faster than JavaIDL.

11 Why do you rewrite the MPI binding but it's still slow?

As an answer to the first question, yes, there have been MPI bindings for Java, such as mpiJava, but they do not suit the need of distributed simulation. Client-server model is not available in MPI 1.x and is only introduced by MPI 2. Bindings that do not support MPI 2 is not useful for the net layers. The best way to get a complete compatible binding for this library is to write one from scratch.

Unfortunately, no MPI implementation is thread-safe until now. Actually, I should say all the implementations have very critical requirements on the threading. The user has to pay a very very high price for the synchronization between C code and Java code, and the synchronization between the only thread that actually calls MPI functions and the other threads. As a result, the layers of synchronous communication and asynchronous communication, which are thread-safe, give poor performance if MPI is the chosen network protocol.

However, this does not forbid the user of net layers to directly call the rewritten MPI binding. It is still very efficient if it is not fit into this thread-safe framework of the higher level.

12 More questions?

For answers of other questions, please email me.