A Capabilities Based Communication Model for High-Performance Distributed Applications: The Open HPC++ Approach

Shridhar Diwan, Dennis Gannon
Department of Computer Science
Indiana University
Bloomington, IN 47401
{ sdiwan, gannon} @cs.indiana.edu

Abstract

Typical high-performance distributed applications consist of clients accessing computational and information resources implemented by remote servers. Different clients may have different requirements for accessing a single server resource. A server resource may also want to provide different kinds of accesses for different clients, depending on factors such as the amount of trust between the server and a client. The requirements or attributes of remote access, such as data compression (and encryption) or client authentication, can be encapsulated under the concept of remote access capabilities.

This paper presents a capabilities based model of communication and describes how it is implemented at application level in a programming environment called Open HPC++. Open HPC++ capabilities are associated with a remote reference and determine the kinds of remote access that it supports. Capabilities can be exchanged between processes, and can also be changed dynamically to helps applications adapt to varying run-time environments. Capabilities also work with the load-balancing features of Open HPC++ to help applications achieve higher performance. The paper presents a set of experiments to demonstrate the usefulness of the capabilities based model. It also describes Open HPC++'s communication protocol adaptivity mechanism, which is the basis of the capabilities based model.

1. Introduction and Motivation

Typical high-performance client-server applications consist of clients accessing computational and information resources implemented by remote servers. Different clients have differing access requirements for a single server resource. Also, a server resource may wish to provide different kinds of accesses to different clients, depending on various factors such as the amount of trust between a client and the server.

Consider a large environmental simulation running on a multi-processor supercomputer at a national lab. There can be many kinds of clients for this simulation. Some clients may only want to get the final weather maps for a geographical area; while some other clients may need to feed data into the simulation and then get the final detailed weather maps. This suggests that while some clients may need access to the complete server interface, others may need access only to a subset of it.

Further, the supercomputer site may want to use authentication for clients connecting over the Internet and would also like to encrypt the data exchanged with such clients. Some clients may be local to the national lab, and so do not need to be authenticated and data exchanged with them does not need to be encrypted.

Some clients may not be associated with the national lab and so they may be given access to the weather data only for the time they have paid for. Some clients may even be given access on a *total number of accesses* basis.

All the above examples suggest that different clients may have totally different requirements of quality of service (QoS), access restrictions, and communication features (attributes), which may be totally unrelated to the specific protocol used for communication. Some of these access requirements can be collected under the concept of *remote access capabilities* (or simply *capabilities*). Once we encapsulate the remote access requirements inside capabilities, we can expect other features also, such as being able to send capabilities from one client to another. Clients should also be able to use capabilities in an adaptive manner depending on the client's and server's environments.

In this paper, we describe our initial experiments with a capabilities based model for accessing distributed server resources, implemented at *application level* in a system

called **Open HPC++**. Open HPC++ is a programming environment for building high-performance distributed applications. Open HPC++ is modeled on the lines of the Common Object Request Broker Architecture (CORBA) [6] to implement seamless communication between distributed heterogeneous components. But unlike CORBA, Open HPC++ does not completely hide its communication mechanism from its applications. It uses the principle of *Open Implementation*[4] to let its applications control its critical internal decisions regarding remote access, while at the same time shielding them from the details of the communication mechanism. Using this principle it supports *protocol adaptivity* features such as multiple, possibly custom, communication protocols, as well as automatic run-time protocol selection with optional user control. An extension of the protocol adaptivity mechanism is used to implement the concept of capabilities. Open HPC++ provides a flexible environment for using capabilities, which includes the ability to use different capabilities to access a server object, the ability to pass capabilities between processes, and also the ability to use capabilities adaptively. Capabilities can also work in tandem with Open HPC++'s load-balancing features to help applications achieve higher performance.

In brief, this paper makes the following contributions:

- Describes a capabilities based model and its adaptivity features for accessing high-performance distributed resources and explains how such a model can be implemented using an *open* architecture.
- Describes how adaptive utilization of capabilities coupled with load-balancing aspects of Open HPC++ can lead to better overall application performance.
- Presents examples and experiments that demonstrate the utility of the capabilities based remote access model.

2. Open HPC++ Design Philosophy

The design of the Open HPC++ communication model is based on the communication models of HPC++ (High-Performance C++) [3] and CORBA.

HPC++ is a C++ library being developed by the HPC++ consortium as a standard model for portable parallel programming. HPC++ provides three main abstractions to facilitate remote access in distributed systems. A *context* refers to a virtual address space; a *node* is a hardware compute resource; and a *global pointer* is a generalization of the `C` pointer type to support pointers to objects residing in remote contexts. It is closely linked to the idea of a remote object reference that acts as a proxy for a remote object. HPC++, by design, supports a restricted model of communication, i.e. it does not support any level of protocol adaptivity.

While HPC++ provides a communication and execution model for high-performance parallel applications, the Common Object Request Broker Architecture (CORBA), has made it possible to seamlessly integrate heterogeneous distributed objects within one system. The CORBA architecture encapsulates its communication mechanism inside a black box, the CORBA Object Request Broker (ORB), thus making application components independent of the actual communication mechanism.

The aim of the Open HPC++ design is to implement the HPC++ global pointer and context abstractions on top of a CORBA-like ORB. But instead of using a CORBA style *closed* ORB, Open HPC++ uses the principle of *Open Implementation* to design an *open* ORB that lets its applications control its critical communication protocol decisions in a limited scope, while still hiding low-level details of the communication mechanism.

Next we see how Open HPC++ enhances the HPC++ abstractions in order to build an *open* ORB. The *open* ORB is used to support communication adaptivity and subsequently the capabilities based model of communication.

3. Communication Adaptivity in Open HPC++

Modern communication libraries like Nexus [2] provide adaptivity features such as multi-method communication, automatic and user controlled protocol selection, and custom protocols. But these libraries provide these features at a very low-level and hence they are not directly usable by application programmers. Open HPC++ attempts to provide similar features at application level. The Open HPC++ ORB, apart from enhancing some HPC++ abstractions, also adds a few new abstractions to make the communication model more flexible (the design of some of these features was influenced by that of Nexus).

3.1. Open HPC++ ORB Abstractions to Support Communication Adaptivity

- *Object Reference (OR)*:
  An OR uniquely identifies an Open HPC++ server object. It contains a table of protocols and protocol specific information (proto-data) that can be used to access the object. The protocols in the OR are ordered by preference.
- *Open HPC++ Global Pointer* (GP):
  An Open HPC++ GP contains an OR representing a remote server object. As different GPs to a single server object may contain ORs with different protocol tables,

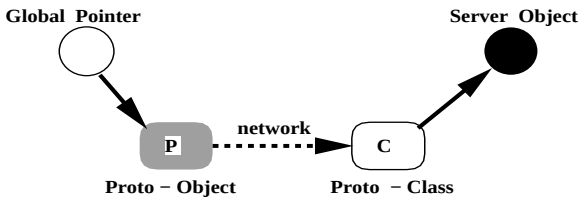


Figure 1. Open HPC++ ORB Communication Mechanism.

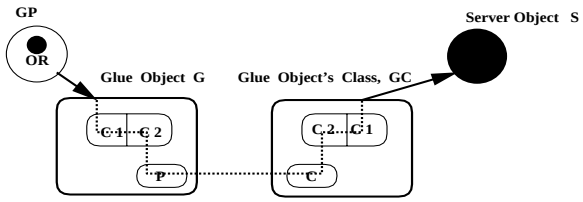


Figure 2. A Remote Request Using Capabilities.

the GPs may support different communication protocols.

- *Protocol Object (proto-object)*:
  A proto-object encapsulates a specific communication protocol; e.g. there could be a TCP based proto-object that uses XDR for data encoding (a proto-object is an instance of a *proto-class*.)
- *Protocol Object Pool (proto-pool)*:
  A proto-pool is a repository of proto-objects, where the proto-objects are ordered by preference. An application component uses a proto-pool to determine the protocols available to it for communication.

3.2. Various Aspects of Communication Adaptivity

This subsection briefly describes how Open HPC++ uses the above abstractions to implement adaptive utilization of communication resources. Please refer to [1] for a detailed discussion of the same.

First let us see how the ORB uses the proto-objects and proto-classes to send a request from a GP to a server object [Figure 1]. When a method is invoked on the GP, the request is sent to a protocol object `P`, which sends it to the server side, and is picked up by the protocol object's class `C`. `C` sends the request to the server object and an appropriate method is invoked. Any return value or exception is sent back to the client in the reverse direction. Open HPC++ ensures that no extra data copying is done over and above that done by the proto-object's protocol implementation.

There are four main aspects of Open HPC++'s communication adaptivity. Firstly, *multiple concurrent protocols* are handled simply by using multiple proto-objects and proto-classes. Secondly, *custom protocols* are supported by having users write their own proto-classes that satisfy a standard interface.

The third aspect is *automatic run-time protocol selection*, which constitutes the ability to automatically select the most suitable protocol for a remote request. In Open HPC++, as different GPs to the same server object contain different ORs, they can access the same server object differently. The proto-object associated with a GP can be changed at run-time thus changing its communication protocol. The system selects an appropriate proto-object for each individual remote request. The OR has a list of protocols that the server object is willing to support. The GP also has a local proto-pool associated with it, that indicates the protocols the local system allows it to use. When a remote request is made, the protocols in the GP's OR are compared with those in the proto-pool and the first match is used to satisfy the request. Thus, the most suitable protocol is always selected.

The fourth aspect is *user control over the protocol selection process*. As described earlier, for a remote method invocation on a GP, the GP's OR and its proto-pool together determine the protocol chosen. Thus, an application can influence the protocol selection decisions by choosing proper ORs and proto-pools.

4. Remote Access Capabilities

4.1. Open HPC++ ORB Abstractions to Support Capabilities

- *Capability Object* (capab-object):
  A capability object encapsulates a specific remote access capability.
- *Glue Protocol Object* (glue object):
  A glue protocol object is a special kind of protocol object that can be used to hold capab-objects in a specific order. The capab-objects are registered with the glue object and they process each remote request before it goes out on the network. A glue object does not contain any communication mechanism but depends on a real protocol object to do the actual communication.

4.2. Implementation of Capabilities

Figure 2 shows the application's view of capabilities. Server object `S` is being accessed by a client through global pointer `G`. `G`'s OR has a protocol table with a glue protocol consisting of two capabilities. The capabilities are:

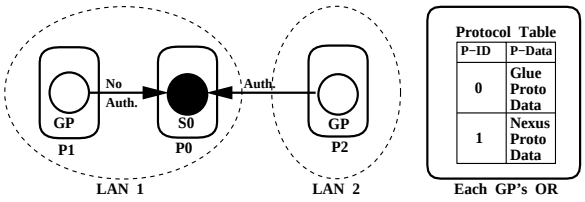
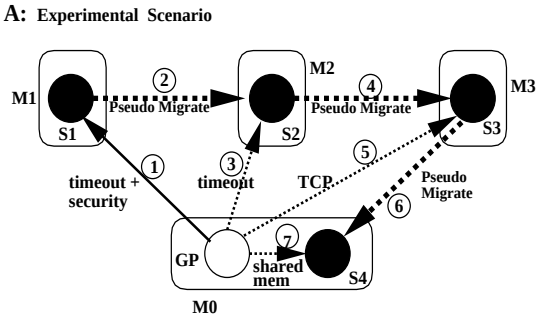


Figure 3. A Sample Application Scenario using Capabilities.

C1, a capability that encrypts the data transferred between the client and the server; and C2, a timeout capability that lets the client make only a certain maximum number of requests. When a client invokes a method on the GP, it is transparently sent to the glue object G. Before putting the request on the wire, the glue object lets the two capabilities process the request. Next, the request is sent to an actual communication protocol object P which sends the request to the server side using its built-in communication protocol. On the server side, this request is picked up by protocol object P's class, i.e. class C. C forwards the request to GC, the class of the glue object G. GC has its own copies of the capabilities, and lets them *un-process* the request in the reverse order of the processing done on the client side. After unprocessing, the request is sent to the actual server object. Any return value from the server object to the GP follows the same path back.

4.3. Adaptive utilization of capabilities

Applications can use capabilities along with other Open HPC++ mechanisms to adapt to varying run-time requirements. Consider Figure 3. Server object S0 is being accessed by two client processes P1 and P2. Assume that the server object requires all clients accessing it from outside its LAN to authenticate themselves for each remote request; while it lets local clients to access its resources without any authentication. The server provides both the clients with copies of a GP whose OR has two protocols, a simple Nexus based communication protocol, and a glue protocol that consists of a single capability implementing authentication, with preference given to the latter. All of Open HPC++'s communication protocols, including the glue protocol, have an *applicability* attribute. Before choosing a protocol for communication, the system checks its *applicability* for a particular connection. For example, a shared memory based protocol is *applicable* only for clients and servers running on the same machine. The applicability of a glue protocol is the logical AND of all its constituent capabilities. The authentication capability can be implemented so that it is *applicable* only when the client and the server



B: The OR's Protocol Table

Proto ID	Proto Data
12	glue protocol with timeout and security capabilities
0	glue protocol with timeout capability
1	shared memory based protocol
6	Nexus based protocol that uses TCP

Figure 4. A: Experimental setup for using capabilities. B: Protocol table for the OR in the GP.

are on different LANs. So, although both P1 and P2 get the same GP for the server object, P1 will always choose the Nexus based protocol, while P2 will always choose the glue protocol with the authentication capability.

Now let us see how Open HPC++ adapts to dynamic changes to the above scenario caused by object migration. Open HPC++ provides a facility for objects to migrate from one context to another [1]. Consider that the load on the server's machine increases beyond a high-water mark and the application decides to migrate S0 to a machine residing on the LAN of client P2. Now P2 becomes a local client while P1 becomes a remote client. For P2, the authentication capability becomes non-applicable, and so it chooses the Nexus based protocol; while for P1, the authentication capability is now applicable and the glue protocol is chosen thus leading to authenticated communication.

5. Experimental Results

This section describes a set of experiments to show how a client can use capabilities adaptively, as its server object migrates from one machine to another [Figure 4-A]. The client context is running on machine M0. Server object S1 starts running on machine M1 and a GP is created for it. The GP's OR has a set of protocols as shown in Figure 4-B. Now assume that between the client and the server, the glue protocol with two capabilities is *applicable*. As it is the first one in the GP's OR, it is chosen for communication. The

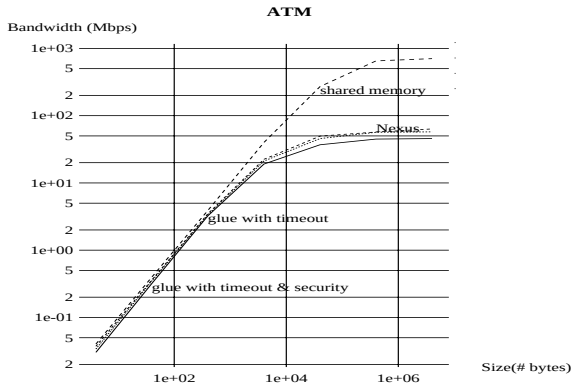


Figure 5. Plot of bandwidth Vs. array size for experiments run over 155 Mbps ATM.

client acquires this GP and makes a series of remote service requests (1). The requests exchange an array of integers between the client and the server, and the average bandwidth over a large number of readings is computed. The requests are repeated for array sizes ranging from 1 to 1 million. Next, S1 migrates to S2 (2). Now assume that between the client's machine and machine M2, the glue protocol with two capabilities is not applicable (probably because they lie on the same campus and so do not need to use secure communication). So the next protocol in the table, i.e. the glue protocol with timeout is chosen and used to make the same set of requests as earlier (3). Next, S2 migrates to S3 (4). Assume that between the client's machine and the machine M3, none of the capabilities are applicable. So, the next choice is the shared memory protocol. But as the client and server are on different machines, the shared memory protocol is also non-applicable. Thus, the system uses the Nexus based TCP protocol (5). Finally, S3 migrates to S4 on machine M0 (6). Here, as the shared memory protocol becomes applicable, it is used to make the requests (7).

The experiments were conducted on a set of Sun Sparc Ultra-10 workstations running SunOS 5.6, connected by Ethernet and 155 Mbps ATM. The results for ATM are shown in Figure 5 (those for Ethernet are virtually identical). In both sets of experiments, all protocols except for the shared memory protocol perform almost identically. This shows that for requests going over the network, the network overhead dominates the overhead of capabilities. Thus it is safe to infer that, even for fast networks such as ATM, the capabilities based approach adds only a small amount of overhead. Also, the shared memory protocol is more than an order of magnitude faster than other protocols thus suggesting that protocol adaptivity can lead to huge performance improvements for dynamic distributed systems with object migration.

6. Related Work

Many communication libraries such as Nexus [2] support protocol adaptivity at the lower level of run-time systems. Nexus also supports communication attributes that can be associated with communication startpoints. Although these attributes provide a facility similar to Open HPC++ capabilities, the letter are more general purpose and also work at the application level.

The Object Infrastructure Project (OIP) [5] has developed a system based on CORBA that supports a mechanism called *illities* similar to Open HPC++ capabilities. The main difference between illities and capabilities is that illities are associated with a piece of code (a thread) while capabilities are associated with a communication endpoint. Thus, compared to the illities, it is easier for capabilities to be passed among processes.

7. Conclusion

This paper presented Open HPC++, a programming environment for building high performance distributed applications. Open HPC++ helps applications utilize their runtime environment adaptively by supporting features such as communication protocol adaptivity, remote access capabilities and dynamic load balancing. The paper shows how Open HPC++ uses the principle of Open Implementation to implement communication protocol adaptivity and remote access capabilities. The experimental results suggest that, capabilities and protocol adaptivity used in conjunction with the load-balancing aspects of Open HPC++, can lead to extremely flexible high-performance applications.

References

[1] S. Diwan and D. Gannon. Adaptive Utilization of Communication and Computational Resources in High-Performance Distributed Systems: The EMOP Approach. In *Proceedings of the IEEE International Symposium on High Performance Distributed Computing (HPDC)*, July 1998.

[2] I. Foster, C. Kesselman, and S. Tuecke. Multimethod Communication for High-Performance Metacomputing Applications. In *Proceedings of Supercomputing*, November 1996.

[3] D. Gannon, P. Beckman, E. Johnson, and T. Green. *Compilation Issues on Distributed Memory Systems*, chapter 3: HPC++ and the HPC++Lib Toolkit. Springer-Verlag, 1997.

[4] G. Kiczales. Open Implementation Home Page, May 1997. http://www.parc.xerox.com/spl/projects/oi/.

[5] Microelectronics and Computer Technology Corporation. The Object Infrastructure Project Web Page. http://www.mcc.com/projects/oip/.

[6] Object Management Group. *CORBA/IIOP 2.2 Specification*, February 1998.