A Multi-protocol Communication Architecture for Metacomputing

FRANCO FRATTOLILLO, SALVATORE D’ONOFRIO
Research Centre on Software Technology
Department of Engineering, University of Sannio
Corso Garibaldi, n. 107, Benevento
ITALY

Abstract: Many current software infrastructures can exploit computing resources accessible via Internet to build computational grids or metacomputers, which can be used both to solve large-scale problems and to access distributed computing resources at low cost. However, computational grids and metacomputers are often characterized by a high level of heterogeneity with respect to the architecture of both computing nodes and interconnection networks. This paper describes the design of a multi-protocol communication architecture for metacomputing applications in Java. The paper analyses aspects regarding both the dynamic integration of different protocols in a metacomputer and the design of communication modules that can reduce the overhead tied to critical paths of communicating Java objects.

Key–Words: Multi-protocol communication architecture, metacomputing, communication libraries

1 Introduction

Exploiting workstation clusters as high performance/cost ratio computing platforms can be considered a clever approach to solve large-scale problems [1], since such exploitation is facilitated by the availability of high-bandwidth, low-latency interconnection systems, such as Myrinet [2], the “de facto” standardization of widely used parallel programming software, such as PVM and MPI, and the implementation of high performance software for managing low-level communication on dedicated network architectures, such as Fast Messages (FM) [3].

On the other hand, Java has become a widely used programming language for distributed computing, since it can make the developed programs independent of the target hw/sw platforms, and directly supports dynamicity, multithreading and security. As a consequence, it has been exploited in many research projects to develop “middlewares” able to harness computing resources accessible via Internet in order to build “computational grids” or “metacomputers” to solve large-scale problems [4, 5, 6, 7].

However, the metacomputers built according to the described approach are often characterized by a high level of heterogeneity with respect to the characteristics of both computing nodes (hardware and operating systems) and interconnection networks. In fact, such heterogeneity can be only partially hidden at application level by using Java, since this language makes the application software independent of the hw/sw characteristics of the exploited computing nodes. However, Java cannot solve the problems tied to the heterogeneity of the interconnection systems.

The problems of the heterogeneity of the interconnection systems exploited within computational grids have been prevalently solved by using IP as a unifying protocol for such systems. In fact, such a solution needs an IP implementation on each network architecture potentially usable in building computational grids. However, since IP is the “de-facto” standard on the Internet, its implementations are commonly available for both local and geographic conventional networks. On the contrary, problems can arise when network architectures specialized for high performance calculus, such as Myrinet, are used. In fact, such architectures are usually exploited by implementing specific, high performance communication libraries in the user space of the operating system of the computing nodes, in order to limit both data copies and overhead due to system calls. Therefore, since all IP implementations are integrated in the operating system kernel, any implementation solution of IP on optimized communication libraries exploiting high performance network architectures would end up requiring data to be transferred from the user space to the kernel one and vice versa. This could cause a performance degradation which might make the exploitation of specialized interconnection hardware useless.

A different approach consists in making it pos-
possible to exploit the communication libraries that are available on the different local area network architectures on geographic scale by means of a unifying interface at application level, i.e. at the user space level. Thus, intra-LAN communication can exploit efficient communication libraries, whereas inter-LAN communication can take place by means of a more conventional protocol, such as IP. Furthermore, it is worth noting that the choice of the protocols to be exploited may be tied to reasons of pure efficiency as well as to application requirements. In fact, grid computing applications are often characterized by components that need to communicate with different requirements with respect to the quality of service. Therefore, even in presence of a single type of physical network, it could be necessary to select different protocols for the different pairs of nodes making up a computational grid or a metacomputer.

Based on these considerations, the paper describes the design of a multi-protocol software communication architecture for metacomputing applications in Java. The paper analyses aspects regarding both the dynamic integration of different protocols in a metacomputer and the design of communication modules able to reduce the overhead tied to critical paths of communicating Java objects.

The paper is structured as follows. Section 2 describes the multi-protocol architecture. Section 3 presents a preliminary implementation of different transport modules: TCP, UDP and FM. In section 4 some performance figures are discussed. Section 5 reports conclusion remarks.

2 The Software Architecture

This section presents the architecture of the software layer integrated into HiMM and JMWare, respectively a Java middleware and a programming framework which enable multi-protocol communications within heterogeneous network interconnection systems [8, 9].

Both HiMM and JMWare implement flexible programming environments based on customizable object-based programming models, such as the “Send/Receive” model or the “Active Objects” model [8, 10, 11]. In particular, applications developed for HiMM or JMWare run on a virtual machine, called “parallel abstract machine” (PAM), made up of a virtual network of processes, each of which is an abstract computing node whose actual processor is the Java Virtual Machine. Each node is allocated onto a host, which can be a PC, workstation or a computing unit of a parallel system, and a host can run more than one node.
Each node of the PAM is provided with two communication modules. The former implements system communications. It is based on a reliable, stream-oriented communication protocol and takes charge of managing the dynamic configuration of the PAM. The latter implements user communications, and is based on a multi-protocol communication interface whose implementations have to ensure reliability to communications. In fact, this approach logically disjoins system communications from application ones, thus making it possible to build computational grids provided with two separated physical communication networks: a conventional network to implement system communications and a high performance communication network to support application communications.

Each node is structured on three layers [8, 9] according to what shown in Figure 1.

The higher layer includes the software modules enabling application programming according to the programming model loaded by the user at the HiM M or JMW are start-up.

The intermediate layer implements the node software components that manage the PAM and the application objects sent/received to/from the network. In particular, objects can be sent to the other nodes of the PAM by invoking the primitives implemented by the components belonging to this layer. On the contrary, the objects received by the node are directly managed by a specific component, called network consumer, defined as a Java interface and whose implementations enable different programming models to be supported at application level.

At the lower level, the network consumer is directly controlled by the underlying transport modules, which are all implemented through the common interface TransportManager (see Figure 2), which abstracts from the details of the actually employed protocols and provides a set of primitives for the unidirectional communication. In fact, this interface has to be exploited to implement transport modules based on new communication protocols, which thus result in being easily integrable into the PAM software support.

The design choice of adopting a unidirectional communication model in defining the common interface of the transport layer is essentially motivated by the programming model exploited by default at the higher level, which is based on “one-way” communications [8, 10]. However, this choice does not limit the possibilities of the multi-protocol architecture, which can adopt a send/receive, message passing semantics by simply installing a customized network consumer provided with “ad hoc” receiving primitives, as reported in [8].

Each node is assumed to be identified by an integer value varying in the range [0, size − 1], where size represents the number of the nodes of the PAM. This enables user applications to refer to the nodes of the PAM independently of the underlying exploited communication protocols. However, such a uniform view of the PAM cannot be preserved at the transport layer, where the transport addresses have to be made explicit. Therefore, in order to correctly de-couple the application level from the transport layer, the interface TransportAddress has been provided. In particular, this interface has to be implemented for each transport module that is to be loaded by the multi-protocol architecture. In fact, when the PAM is created at run-time, coherent transport addresses are assigned to each dynamically loaded implementation (for example, IP address and port number for UDP and TCP). More precisely, during the start-up phase of the PAM, each node can know the set of the protocols that have to be loaded by reading a specific configuration file. Then, each node can send the other nodes the transport addresses of the loaded communication modules, which can be dynamically retrieved by invoking the primitive transportGetAddress belonging to the interface TransportManager. As a consequence, each node, once it has received a transport address, loads it in a data structure, called endpoint, which locally represents the corresponding remote node. Thus, each node stores an array of endpoints, whose size is equal to the number of nodes of the PAM minus one.

The end points interact with the net-

```java
public interface TransportManager {
    public TransportAddress transportGetAddress();
    public void transportSend(TransportAddress, Object);
    public void transportMulticast(TransportGroup, Object);
    public void transportBroadcast(Object);
    public void transportConfigure(Parameters);
    public void transportClose();
}
```

Figure 2: The common interface of the transport layer.
work layer through the implementation of TransportManager. Therefore, when the running application wants to send an object to a given node of the PAM, the integer value identifying the target node allows the corresponding end point on the sending node to be accessed. Thus, the primitive transportSend belonging to the implementation of the transport module associated to the identified end point can be invoked in order to send the object.

It is worth noting that the primitives of the interface shown in Figure 2 accept Java object as arguments. This requires that each transport module implements a mechanism to marshalling/unmarshalling the exchanged objects. However, the choice of implementing such mechanisms in the transport modules makes it possible to achieve a higher level of efficiency, since the optimal implementation of the marshalling/unmarshalling algorithms closely depends on the adopted communication protocols.

3 The Implementation of the Proposed Architecture

Three transport modules have been implemented for the following communication protocols: Transmission Control Protocol (TCP), User Datagram Protocol (UDP) and Fast Messages (FM). The modules for the first and second protocol have been developed on the basis of the standard support to communications supplied by Java for TCP and UDP, whereas the transport module for FM has been developed by directly exploiting the original communication libraries for this protocol through the Java Native Interface (JNI) technology.

3.1 The TCP module

This module has been implemented by exploiting the TCP socket interface supplied by the standard Java Development Kit (JDK). In particular, to reduce the communication latency for small size messages, the Nagle algorithm has been disabled. In fact, this algorithm enables data to be stored in a buffer before being sent, and this can limit the effects of the start-up costs of a TCP connection when large size messages have to be sent. As a consequence, the round-trip delays obtained by the implemented TCP module usually result in being lower than those ones obtained by the Java standard module. Furthermore, to solve the performance problem caused by the start-up overheads in establishing TCP connections, the implemented module creates persistent TCP connections among all the nodes taking part in a PAM. In particular, each bi-directional connection is created upon the first communication between two nodes. To this end, the primitive transportSend implemented by the TCP module running on the sending node creates a server socket and sends its port number to the TCP module of the target node, whose transport address is known. Once the port number is received, the target node can establish a TCP connection with the sending node by employing the received port number. Then, the server socket can create a new socket connection in order to send data to the target node. In addition, in order to keep the created TCP connections alive as well as to enable the TCP modules of other nodes to establish bi-directional connections, the TCP module of a node exploits a dedicated thread, which is suspended waiting for the requests of generating new TCP connections coming from the other nodes to its transport address. This thread, once resumed by a new connection request coming from a node, creates a new thread, which takes charge of managing the communication with the node.

Establishing persistent connections among the nodes of a PAM can reduce the communication start-up costs. However, it can also limit the system scalability, since operating systems allow for a maximum number of active TCP connections onto a single host. As a consequence, this number becomes a relevant constraint on the maximum number of hosts which are allowed to take part in a PAM.

3.2 The UDP module

This module is based on the UDP socket interface supplied by the standard JDK. However, it also implements a “selective repeat” protocol based on “negative acknowledgement” on top of UDP in order to provide a reliable connectionless delivery service able to transport messages of arbitrary size without forcing users to directly fragment them or to order incoming fragments. Furthermore, this module does not influence the system scalability, in that it does not implement persistent sockets among the nodes of the PAM.

3.3 The FM module

This module exploits a dynamic library developed on the basis of the C communication library implementing “Fast Messages” [3]. This library supports high performance low-level communications on top of a Myrinet interconnection system and by employing a simple programming interface whose semantics is based on the “Active Messages” communication model [12].

The current implementation of this module exploits Java native functions that encapsulate the FM primitives supporting send/receive communications
class FMManager implements TransportManager {
    ...
    <algorithms for marshalling/unmarshalling objects>
    ...
    private native void JFM_send(byte[] msg, int node);
    private native byte[] JFM_receive();
    private native void JFM_init();
    private native void JFM_end();
    private native int JFM_thisNode();
}

Figure 3: The Java interface for the FM send/receive communication primitives.

(see Figure 3). In particular, data that is to be sent is stored into a Java buffer passed on to the JFM_send, which builds a message, joins a specific FM “handler” to it, and sends the message to the target node.

A low priority thread running on the target node continuously polls memory to determine if the sent data is available by invoking the JFM_receive. Then, upon the message receipt, the function JFM_receive invokes the FM primitive FM_extract, which executes the FM handler associated to the message. Thus, message data is stored into a buffer returned by the JFM_receive. Then, the unmarshaling process creates a Java object from the data stored in the buffer, and passes it to the network consumer, which takes charge of manage it.

Figure 4: The round trip delays measured for the implemented communication modules.

4 Preliminary Tests

This Section briefly reports on some performance experiments conducted by exploiting a cluster of PCs connected by a Fast Ethernet and a Myrinet switch. The cluster is composed of 16 PCs equipped with Intel Pentium IV 3 GHz, hard disk EIDE 60 GB, and 1 GB of RAM. All the PCs use the release 5.0 of the SUN JDK.

Figure 4 shows the round trip delays obtained by sending messages of varying size between different nodes according to a simple send/receive communication model. To this end, each value shown in Figure 4 is calculated as the mean over more than 1000 iterations.

The results obtained by the proposed multi-protocol software communication architecture can be considered essentially good, even though the UDP module is affected by a performance penalization due to a not optimized implementation with respect to the TCP module. In fact, the implementation of the UDP module is penalized by the adopted object marshalling/unmarshalling algorithms, which have been developed for a stream-oriented communication style.

A not optimized implementation also characterizes the FM module, which has been developed without exploiting all the potentialities tied to the use of the FM communication library.

5 Conclusions

This paper has presented the design of a multi-protocol software communication architecture for metacomputing applications in Java. The architecture can exploit the communication libraries that are available on the different local area network architectures on geographic scale by means of a unifying interface at application level. Thus, intra-LAN communication can exploit efficient communication libraries, whereas inter-LAN communication can take place by means of the more conventional IP protocol.

The proposed architecture has been also tested by conducting some experimental tests, which have demonstrated that flexibility can be achieved without reducing performance.
References:


