A Real Time Application based Java-Based Distributed Object Processing System for IP Telephony

G Vijay Kumar
Department of CSE
Sri Sunflower College of Engg. And Tech.
Lankapalli, Challapalli, Krishna Dist 521131
Krishna Dist., Andhra Pradesh, India
Ph.No: 08671258585, Cell No: 91-9849748772
E-mail Id: vijay.g.teja@gmail.com

Dr G V Raju
Principal
Sri Sunflower College of Engg. And Tech.
Lankapalli, Challapalli, Krishna Dist 521131
Krishna Dist., Andhra Pradesh, India
Ph.No: 08671258587 Cell No: 91-9849748772
E-mail Id: principal@sunflowercet.org

Abstract

We have developed a Java-Based Distributed Object System, JDOP, to solve the problem of distributing, managing, and interconnecting a set of independent worker objects, which comprise a distributed computation. A JDOP client is aware of the objects required to solve the computation and provides them as well as the necessary data to a JDOP Master Server. The JDOP distribution system is charged with receiving client requests, storing pertinent client information, distributing the objects to the JDOP distributed object execution subsystem, and finally receiving and forwarding the results back to the client. The JDOP system can handle distributed computations similarly to PVM or the MPI standard [10,11]. However it has features, which give our system advantages. First, the system is completely Java-based, which makes it simple to distribute the computation anywhere a Java Virtual Machine (JVM) resides [7,13]. Second the computation is distributed across a dynamic set of nodes without affecting the client. Third, the protocol for JDOP computation is simple for the client, thus making the coding for client programs simple. Lastly, and most importantly, the system is completely client-oriented. The client does not need to access pre-registered services as exist in Java RMI, CORBA, or typical naming services [1,7,13,14].

Keywords: Distributed Objects, RMI Parallel and Distributed Processing, Multi-Tier Systems.

1 Introduction

Distributed application systems allow the exploitation of the existing computing infrastructure to utilize unused CPU cycles [1,4]. A computation is usually broken up into a set of processes, which are assigned and executed on the machines within the network or cluster. Such systems enable independent processors to simultaneously execute the jobs to improve overall completion time. UNIX, and now other platforms offer RPC, or remote procedure call. RPC allows the client to specify a host from a set of trusted hosts on the network to be a target host of a procedure from a process executing on a client host [4]. RPC is dynamic because it does not require the target host to have a priori knowledge of the procedure. However, it is not simple to use, requiring compilation of special interface files, and it is not an object oriented, or platform independent solution. Typical procedural systems such as PVM create a virtual machine from a collection of LAN workstations. New processes are spawned from a driver process. All control and coordination of the processes are done through a set of APIs provided by the system [11]. These APIs are often large and complex, and require a great deal of expertise from the programmer. Object-oriented systems, such as Java RMI or the CORBA standard, work by having object servers registered and available [1,13,14]. Client objects simply perform an object lookup to find the appropriate server object and are returned a remote reference. The clients then use the remote object as if it were a local one, unaware of the distribution. Implementing the servers in such environments requires the software engineers to adhere to appropriate APIs and registration protocols in order to make their servers available to clients. If a client requires a service, which is not currently registered in the naming service, that process cannot be executed. JDOP allows tasks and server objects to be loaded and executed remotely and dynamically.
Our approach in the Java-Based Distributed Object System, JDOP, is to make an ordinary object become a server object dedicated to the client. It enables the JDOP virtual network to use client-defined components, promoting CBD, component-based software development, usage [8]. Further, a client is not aware of the fact that the object will be run non-locally, or its specific location of execution. Our protocol is client-centered. That is, a client determines what objects it needs, and makes a request for any number of instances of any types of objects.

The system architecture is a three-tier model. At the bottom of the hierarchy is the client. Because the architecture is client-driven, the complexity of the client is slightly more complex than a comparable RMI model. The middle tier is the Master Object Server, MOS, which effectively coordinates the set of clients, tasks, and server objects. Last, the Slave Object Server, SOS, is the tier responsible for actually instantiating the server object fetching the result set, and returning it. Figure 1, shows the architecture. The Application Server, AS, allows the whole system to be relocated without affecting the major three components: Client, MOS and SOS.

2 JDOP Architecture

There may be any number of clients sending jobs to a MOS. Each job can be a set of any number of workers. There may be many SOSs, each capable of running worker objects. Figure 2, shows a set of clients, a MOS, a set of SOSs, each with a different number of worker objects. The number of clients, workers, and SOSs will vary over time, allowing the system to scale up as needed.

2.1 The Client

All three tiers are Java-based, and run the JVM locally. As such, all three have access to the standard Java classes. None have any knowledge of any non-standard or special server objects. A client is basically a consumer class, which has one or more tasks to do. Each task is independent, and its implementation is coded in a server (worker) class. The client contacts the MOS with a (worker) class name and a data input set. This class, when instantiated will become the server object or worker. The MOS may be unaware of the class, and will, in such a case, make a request of the client for the Java bytecode defining the class. The client, which is essentially a job submission and result collection program, typically goes through two phases. The job submission phase in which classes and data sets are sent to the MOS for execution, and a collection phase during which the results are collected. Upon each job submission, the client receives...
an task id, \( tid \), from the MOS. The \( tid \) is a unique number, which corresponds in the system to a specific object. When the worker task is completed the client will receive a message in which the \( tid \) is followed by the result set of the computation. The \( tid \) allows the client to differentiate as to which particular worker task result set follows. There is a specific way in which every worker must be defined. First, since it is a object running as a thread inside an SOS, it must be made runnable by implementing the \texttt{Runnable} interface. Second, in order to facilitate the activation of the worker as well as the collection of the result set, the worker adheres to a specific simple \texttt{Worker} interface. This interface is standard allowing any SOS to execute any such conforming worker object.

2.2 The Master Object Server (MOS)

The MOS tier is a true manager and coordinator between the client and SOS layers. Its chief responsibilities are to make the distribution of the workers across the network transparent to the client, and to keep track of the links between the clients and the SOSs. The MOS receives requests from any number of clients submitting any number of worker classes and data sets. Upon receiving a request, the MOS keeps a table filled with the task ID \( (tid) \), task status, as well as data streams to the client and SOS associated with each task. It is the responsibility of the MOS to find an appropriate SOS for the execution of a worker. The MOS is also constantly accepting connections from new SOSs. As such, the size of the distributed machine can scale without affecting a JDOP client or SOS.

The basic MOS implementation is as follows. First a MOS is started on a machine. The MOS makes a POST operation request using HTTP protocol to a fixed URL. In the current implementation that process is using Microsoft active server pages, asp, technology. This registration process makes the MOS through its IP address and port available to both clients and SOSs. A client will contact the URL, using a special client port, which will simply return the location of the MOS port listening for new tasks. A newly started SOS will contact the same URL, with a different port, to get the location of the MOS port listening for registration of new SOSs. Once such a new SOS request is received, the MOS adds this machine to its pool of available SOSs. The SOS also sends its location and port, so that MOS task communication is possible. Once a client new-task request arrives, the MOS simply chooses an SOS based on some busyness criteria. This can be a simple round-robin approach, a CPU process count, or any other heuristic means.

2.3 The Slave Object Server (SOS)

The SOS tier is basically a task manager. Its primary function is to instantiate the worker, invoke the proper \texttt{Worker} interface operation passing the worker its data set, and finally collect the result set from the worker and forward the results to the appropriate MOS. The SOS can become a part of the JDOP system at any time. Whenever a new SOS is created, it connects to an appropriate URL, which return the IP and port of a running MOS. The SOS will then signal the MOS, and register itself as an available server for worker job submissions. Once the MOS with a job request has contacted the SOS, the SOS will run the worker task it receives in a separate thread. The MOS request comes with a bytecode, if necessary, and the task id, \( tid \) the worker has been assigned. The SOS must maintain a table of workers, containing their MOS IP number and \( tid \). Upon completion of the task, the SOS sends the \( tid \) and the accompanying computed result set to the MOS.

3 JDOP Example

In this section, we follow though a run of the JDOP system as it executes workers of a matrix multiplication class. Various screenshots are taken to show the Client, MOS and SOS component at various stages of the process. In Figure 2, the MOS registration process is shown. A MOS server is started. It looks up the AS, and sends to it its own IP and port pair. One for the Clients listening port, the other for the SOS listening port. In a similar manner, the SOS also does an AS lookup to find the registered MOS. Once found, it sends its own IP and port so as to allow the MOS to use this SOS for worker execution. In our example, the Client has requested 4 instances of the worker to be executed. The client performs a matrix multiplication, and breaks the job into four pieces. Each worker will compute a quadrant of the resultant matrix. Figure 3 shows the Client receiving result sets from the MOS. In Figure 4, we see the SOS actually processing one of the matrix multiplication workers. The JTable in the lower SOS panel displays the \( tid \) and the associated computed data streams. The last state indicates that the results are being sent back to the MOS. The \( tid \) is always kept with the state information because the SOS is potentially running multiple workers from various clients.

4 Future Model

The future of JDOP is to allow it to be even more scalable and to implement it more ubiquitously. To these ends, the entire system, specifically the MOS and SOS subsystems will be implemented as web-based services. Our aim is to make the MOS and SOS server-side processes, such as servlets. This is a natural implementation for a Java-based system. The servlet implementation will automatically multithread the JDOP components making them more responsive and less
resource demanding. A MOS or SOS installation would also be trivial, each being a simple servlet install. To scale better, the one MOS in JDOP can be replaced by a set of server-side DMOS modules in the web-based wide, distributed model. This would create a web of potentially unlimited size. To realize this, we would need to improve and well-define the initialization processes for DMOS and SOS modules to make the wide distributed system efficient. The primary areas to address are the balancing and localizing computations in the DMOSs and associating the SOS modules to the DMOS managers. Other enhancements to our system will include connectivity through SOAP, Simple Object Access Protocol [6,12]. This is a lightweight, XML based protocol, which allows web services to be packaged and communicated across the Internet using HTTP and HTTPS. SOAP is a messaging protocol that is not limited to RPC, and does not require synchronous execution [12]. Additionally, SOAP messages can have multiple parts addressed to different parties. We intend to tie JDOP to web services, where some of the workers we spawn could be web services. The most important goal we have is to redesign JDOP for use on systems where there is worker communication. Presently, the workers executing on the SOSs are working on data sets provided. They do the work individually, and return the result to the client. The next phase would be to allow the workers to send messages to other workers executing in parallel on other SOSs. Because the location of the fellow workers is unknown to their peers, the entire messaging system would be transparent to them, thus greatly simplifying the worker design. This work listed is currently in progress.

5 Conclusions

We have developed a simple way to create, run and manage a set of distributed objects across a local or wide area network. Our system, JDOP, is simple to use, and it does not require the server to have registered services. The MOS/SOS servers can dynamically accept classes and data sets, and execute them remotely. Our system is different than Java RMI or CORBA systems as it allows clients without any administrative privileges to order any task they desire. In this way we have developed a client oriented distributed processing model.

6. References


