Abstract

The JabberGrid project investigates ways to use loosely coupled computing resources to perform decomposable work. The primary focus of the project is on using the Jabber/XMPP (Extensible Messaging and Presence Protocol) instant messaging technology as a communications and coordination platform. This technology provides a reliable, secure, store-and-forward network with an extensible protocol.

We have developed a small set of demonstration applications for JabberGrid that solve some trivial problems in parallel on several machines. JabberGrid operates as follows: a user specifies the problem as a 'work package' to the distributed application, which will then consume the problem and parcel out the work to as many workers as needed. The novel use of an instant messaging (IM) system to coordinate agents in a distributed program have revealed technical issues that are common to any distributed problem solving effort, as well as problems unique to .

This paper describes the architecture of JabberGrid, the distributed challenges we addressed in the project, and the type of problems we solved with JabberGrid. We also presents some initial evaluations on performance and scalability.

1. Introduction

The Jabber technology has the potential of being an important keystone technology. The Jabber technology provides a network of servers that provide store-and-forward capability, security and an extensible messaging system, while allowing diverse
programs to use the network for a variety of purposes.

The Jabber Project (http://www.jabber.org) has developed an open-source alternative to the Instant Messaging technology provided by AIM (AOL Instant Messenger) and Yahoo! Messenger. Jabber technology consists of one or more networked servers and a variety of diverse clients which connect to this server network. Typically, these clients are GUI-based and allow different users to 'chat' across a network by typing into a window. Jabber manages the transmission of these messages from one client to another, including a store-and-forward capability for offline clients. Messages are exchanged using XMPP (Extensible Messaging and Presence Protocol), which is an XML-based protocol used on top of a TCP/IP stream. There are many ways Jabber can be exploited in building both middleware and distributed applications. Ultimately, to fit within the project bounds, we narrowed the scope of JabberGrid to be a simple distributed problem solving system that is still sophisticated enough to illustrate some common distributed systems issues as well as providing a testbed for performance.

JabberGrid is motivated by the idea of grid computing. A grid is a collection of distributed and heterogeneous resources that are dynamically discovered and allocated. This allocation in many of today's grid implementations occurs through centralized or decentralized resource brokers or meta-schedulers that connect grid clients requesting a service with providers based on service requirements [1].

Central to the idea of a computing grid is the set of services that coordinates access and use of the grid by providing facilities such as lookup and discovery of available resources, security, and transaction management. Grids can be thought of as a sort of marketplace that brings users (service consumers) and service providers together for the purpose of providing transparent access to a collection of heterogeneous computing resources.

JabberGrid has the potential of being a resource broker that can handle discovery and
coordination of workers and services, distribution and collection of problems and results, and workflow management. From the Jabber technology the project inherited some security and fault tolerance features that we have exploited. Evaluation show that JabberGrid can handle an impressive size of workpacket (2.2 gb), therefore is reasonably scalable, performs as well as Java RMI, and is suitable for non-trivial problems.

This paper expands on these ideas and is organized as follows. Section 2 describes the JabberGrid architecture and the terminology used in describing the architecture. Section 3 describes the problems we solved, which emphasize the issues relating to distributed work systems and then those issues are discussed in Section 4. In Section 5 we describe and analyze the performance and scalability evaluations we performed. Section 6 we briefly discuss future work and then concludes in Section 7.

Preceding this paper is an appendix containing user documentation.

2. Architecture and Terminology of JabberGrid

2.1 Third Party Architecture in JabberGrid

The JabberGrid project builds upon several components that are open-source and available free of charge. These third-party technologies come with their own set of terminology and acronyms, which are important in describing the JabberGrid architecture.

This section describes the third-party technologies and terminology that are relevant to JabberGrid.

2.1.1 Instant Messaging – IM
Instant Messaging (or IM) has been available on computer systems in one form or another since the earliest days of multiuser computers. Early forms of interactive chat were available in the form of IRC (Internet Relay Chat) as well as various 'talk' programs.

One key aspect of Instant Messaging is that it allows one participant to send a small text message to another participant in real-time, knowing that that other participant is online and likely to respond to the message. This allows two participants to Chat with each other by sending messages back and forth, carrying out a conversation.

Another, almost as important, aspect of Instant Messaging is the idea of presence, which refers to the current state of a participant with respect to their Online and DoNotDisturb status. The presence mechanism of a good IM system allows a potential sender to discover which potential recipient is currently online. In addition, it allows a participant to indicate their intended status (e.g., DoNotDisturbExceptInEmergency).

**2.1.2 Person-to-Person vs Chat Rooms**

Basic IM allows point-to-point conversation between two users or agents. Building on this basic Chat idea, the idea of MultiUserChat was developed. This allowed multiple participants to be aware of each other’s presence in a Chat Room and to share messages with the other participants in the room. Although the Room metaphor is not required to allow these types of multiparty conversations, it is quite common in IM systems. Other alternative names for this type of multi-party conversation include Conference or Discussion.

JabberGrid relies upon MultiUserChat to implement a multicast protocol amongst the participant workers in a room. We will use the terms MultiUserChat and Chat Rooms interchangeably, although the former actually refers to the Smack Java class and the latter refers to the abstract concept of a network "place" where messages can be shared
amongst the room participants.

2.1.3 Jabber

Jabber is the umbrella name for the set of protocols (XMPP), standards (RFC 3920, et al), server software (jabberd, JiveMessenger), and client software (Gaim, Smack) that is used to provide IM(Instant Messaging) services. Jabber is an open-source alternative to proprietary Instant Messaging systems such as AIM (AOL Instant Messenger) and Yahoo Instant Messenger.

A quote from the http://www.jabber.org website provides this definition:

Jabber is best known as "the Linux of instant messaging" – an open, secure, ad-free alternative to consumer IM services like AIM, ICQ, MSN, and Yahoo (see the IM quick-start). Under the hood, Jabber is a set of streaming XML protocols and technologies that enable any two entities on the Internet to exchange messages, presence, and other structured information in close to real time.

There are a wide variety of Jabber-capable clients and servers, many of them free and open-source.

More information on the Jabber technology is available at: http://www.jabber.org.

2.1.4 XMPP (Extensible Messaging and Presence Protocol)

More information on these RFCs is available at http://www.xmpp.org.

XMPP uses TCP/IP connections to transmit XML messages between XMPP clients and XMPP servers, or between multiple servers. This capability of networking multiple servers and their clients makes this technology attractive for certain distributed problems.

XMPP specifies that a client opens up a connection (TCP/IP is the only one in common use) to a server. This bidirectional connection can support the transmission of messages known as stanzas.

2.1.5 XML Stanzas

The messages that are used in XMPP are encoded as proper subtrees of XML. These XML stanzas act as messages in a message passing system. Each stanza contains structured information including addressing and routing information, encryption, and arbitrary amounts of structured payload data. The routing information is used by the XMPP server to relay the stanza to one or more clients, possibly by routing through different servers.

2.1.6 XMPP Packet Extensions

The X in XMPP stands for extensible. Packet Extensions are a way to embed arbitrary XML payloads within certain XMPP messages (Presence and Chat). A packet extension is uniquely identified with an element name and a namespace. The namespace is typically a string of the form 'jabbergrid:x:WordWorker', and is used by XMPP endpoints to extract relevant packet extensions from an incoming message.

2.1.7 XMPP Server
An XMPP server is a program that can manage multiple concurrent connections from clients that communicate using XMPP. XMPP servers are also capable of communicating via Server-to-Server communication. In this way, XMPP messages can be routed to far-remote clients by delegating the message to the remote client’s local XMPP server. In addition to supporting client connections and relaying messages (XML stanzas), most XMPP servers provide services such as maintaining a persistent store of messages, so that in the event of a network connection breaking down, messages may be delivered after the network comes back up (similar to a store-and-forward email system).

2.1.7.1 jabberd

jabberd is an open-source XMPP server. It is a very robust server that requires complex administration and database setup. We will not be using it for this project, although it should work fine, since the XMPP protocol is the same in both cases.

2.1.7.2 JiveMessenger

JiveMessenger is a particular XMPP server that is provided by Jive Software (http://www.jivesoftware.com) free and open-sourced at http://www.jivesoftware.org/messenger/.

JabberGrid will use JiveMessenger during its development and evaluation.

2.1.9 XMPP Clients and Client Frameworks

2.1.9.1 GAIM

GAIM (http://gaim.sourceforge.net) is an open-source, multiprotocol, IM client. It is
capable of communicating via the Jabber protocol and is quite up-to-date with respect to the current XMPP standard and its extensions.

2.1.9.2 Smack Framework

Smack is a set of Java classes that provide a convenient way to write client programs that use XMPP to communicate with an XMPP network. A client author using Smack can program at the logical level of users, chat rooms, and messages. Programming clients directly with TCP/IP and raw XML is possible, but it would require much more effort. The JabberGrid project will use Smack to access the XMPP network. Smack is provided by Jive Software (http://www.jivesoftware.com) free and open-source at http://www.jivesoftware.org/smack/.

2.1.10 JID or JabberId

A JID is a string which identifies an XMPP client. An example of a JID is:

    worker0@127.0.0.1/WordTask

which refers to a login (either human or programmatic) named 'worker0' on an XMPP server at 127.0.0.1 with resource 'WordTask'. The optional Resource qualifier allows disambiguation of clients that connect using a common login name (e.g., 'worker0'). This is used by JabberGrid to allow a small number of registered XMPP users to support an arbitrary number of Workers and Tasks.

2.1.11 Rooms are an illusion

An XMPP client (with appropriate permissions) create a new MultiUserChat room. This room is identified with a name (e.g., WordTaskRoom) and a host (e.g., conference.127.0.0.1). Participants who are logged into an XMPP server can either
open a Chat session with an individual (Instant Messaging), or they can open a MultiUserChat session with a MultiUserChat session (i.e., a ChatRoom).

In either case, the network connection is always between the client and its local XMPP server. There is no point-to-point communication between clients. In the case of a MultiUserChat, a message sent into the room via one participant is automatically replicated (multicast) to all of the participants in the room by the XMPP server.

This automatic multicasting of messages to a room's participants is what creates the illusion of a shared space, usually called a room. The MultiUserChat extensions to XMPP also support the ability to discover what participants are also connected to the room (as well as ways to eject participants). This adds to the illusion that is a chat room.

2.2 JabberGrid Architecture

JabberGrid is a framework of Java classes and utilities that rely upon an existing network of XMPP servers to communicate. Program authors can use the JabberGrid toolkit to build a variety of distributed applications. These applications typically consist of a coordinated set of Task and Worker objects that run within various processes. As more nodes are added to the network, so may more Workers be added, increasing the problem-solving power of the grid.

2.2.1 JabberGrid

A JabberGrid object encapsulates a single XMPP connection and provides some useful abstractions such as queuing and conversion to/from WorkPackets.

One of the tricky problems we looked at in JabberGrid is making a version of JabberGrid that supports a SmartMulticast that guarantees that a message issued to multiple users is only 'accepted' by one. In effect, the possible recipients of a message
use a ‘consensus’ algorithm to select a unique chosen recipient. We believe that other, traditional, consensus schemes could be applied here also.

### 2.2.2 Workers and Tasks

The JabberGrid programming framework allows the creation of subclasses of Worker and Task. A Task is responsible for creating a room (optional), injecting input messages into the room, and possibly waiting for interesting or useful messages to appear in the room. A Worker listens for work requests and carries them out when they are received. In some cases, a Worker will listen for messages from a room. In all cases, a Worker may receive messages directly.

The class Task is actually a subclass of Worker, with additional methods to allow the creation of a room. Tasks are Workers that host a particular Room. One of the ways to use a Task is to have it create a room, inject a question, and wait for an answer to appear.

### 2.2.3 WorkPacket, Header, RoutingSlip, Data

This is a structured Message that uses XMPP Packet Extensions to encode a variety of submessages in the form of Headers. A Header can record name/value pairs, as well as providing routing information. In addition, a Header has a nextHeader field which is used to create an ordering of Headers within a WorkPacket.

There are two subclasses of Header that are also used (in addition to Header itself): Data and RoutingSlip. Each WorkPacket has a single RoutingSlip header and a single Data header. The RoutingSlip header serves two purposes: it indicates that this Message is in fact a WorkPacket, and it indicates which Header should be executed next in this WorkPacket. The Data header stores the mutable payload for the
2.3 WorkPacket Structure

Figure 1 shows how a WorkPacket is composed of multiple Packet Extensions.

2.3.1 Packaging of the JabberGrid Demo

All of the components (classes) needed for the various processes in the JabberGrid demo are packaged as a single Java application jar file called JabberGrid.jar. Different
functions are accessible as command line arguments such as “wordworkers”, “workerpool”, "init", and so forth. These different commands to the single JabberGrid program are referred to as “commands” in the following description, even though are simply subfunctions of the JabberGrid main program.

Using the “ant” command is the easiest way to run these different commands/programs. For example, the following command conveniently runs the JabberGrid program and passes it the ‘wordworkers’ command:

> ant wordworkers

2.3.2 JabberGrid Software Layers

Figure 2 shows the different layers used in the JabberGrid Demo Program.
2.3.3 JabberGrid Network Layout

Figure 3 shows how Workers, Tasks, and IM clients all share access to the same XMPP Network "cloud".

2.4 Control and Message Flows within a JabberGrid Program

The above static description of the components of the JabberGrid is only part of the background necessary to understand JabberGrid. In this section, we will describe the significant messages and events that characterize a typical JabberGrid distributed program.

We will frame this description in terms of the WordTask/WordWorkers demonstration from the JabberGrid prototype. This trivial distributed application consists of two processes. The first process, called WordTask, uses the WordTask class to create an XMPP room, which it populates.
with WorkPackets describing texts to be transformed. In the current demo, the transformation is simply uppercasing the text; we envision more complex transformations such as language translation would be a more realistic demonstration of JabberGrid’s potential.

A second process, called WordWorkers, runs one or more WordWorkers as separate threads within this process. These WordWorkers (running in separate processes from the WordTask) wait for the appropriate WorkPackets to appear in the room, at which point they consume the tasks, carry out the work, and reply to the WordTask which initiated the work.

Note that practical, non-trivial, distributed programs using JabberGrid will likely have a more sophisticated structure with more Rooms, Workers, and Tasks. However, the example of uniform Workers receiving work from a single Task is useful and easy to understand, so we will begin with this example, and then consider possible extensions to the example.

2.4.1 One time initialization of XMPP Server

Before running any JabberGrid demonstration programs, it is necessary to create authorized logins within the XMPP server to allow the demonstration to work. The JabberGrid program has an “init” command-line argument which causes it to create the required users in the XMPP server’s database. This command need only be run once per XMPP installation.

2.4.2 Start “wordtask” process

The “wordtask” command creates a JabberGrid which logs in to the local XMPP server as user “wordtask”, and a WordTask class to be the problem coordinator.

This WordTask class creates a room on the XMPP server and joins the room. This step
is common to most types of Task. The Task subclass creates one or more task-specific rooms and joins the room in order to inject work requests and receive completed work.

2.4.3 WordTask composes WorkPackets and injects them into room

In the WordTask example, the WordTask has been given a list of texts to be processed by distributed workers. For our purposes, suppose these texts are trivial: “the”, “quick”, “brown”, and “fox”.

The WordTask will construct (for each of these texts) a WorkPacket which contains the following information:

• A RoutingSlip, which names the current workflow step to be performed (“Step1.Uppercase”, in our example).
• A Data object, which contains the payload to be worked on (the text, in this example)
• A WorkFlowStep object named “Step1.Uppercase”, with a workType of “WordWorker”
• A WorkFlowStep object named “Step2.Collect”, with a workType of “WordTask”

This WorkPacket describes a two-step WorkFlow that is currently positioned at the first step, “Step1.Uppercase”.

2.4.4 Start “wordworkers” process

The “wordworkers” command runs a process that creates a WordWorkerFactory, which creates one or more threads running WordWorker objects.

2.4.5 Each WordWorker joins WordTaskRoom
As each WordWorker runs, it will join the WordTaskRoom, whose name is passed to the WordWorker when it is created.

2.4.6 WordWorker waits for WorkPackets

Once a Worker joins a room, all messages that have been inserted into the room since its creation are available to be delivered to the Worker. By default, JabberGrid Workers use the SmartMulticast feature, which allows the participants of a room to negotiate which worker will actually receive a given WorkPacket. This ensures that distinct WorkPackets will be delivered to each worker in a room.

2.4.7 WordWorker processes a WorkPacket

When a WordWorker’s deliverWorkPacket() method is called, the WordWorker can be assured that the JabberGrid SmartMulticast has negotiated with the other workers and that this worker has exclusive access to the WorkPacket.

The WordWorker will examine the packet’s RoutingSlip and will determine that the instruction to be carried out is “Step1.Uppercase”. This WorkFlowStep indicates that the payload text should be uppercased and reinserted back into the WorkPacket. The WordWorker performs this function, inserts the transformed text back into the WorkPacket, and then invokes WorkPacket.advanceStep(), which adjusts the RoutingSlip to point to the next WorkFlowStep.

In this case, the next WorkFlowStep (“Step2.Collect”) indicates that it should be processed by a different entity, in this case, the WordTask. This is indicated by the WorkFlowStep having a "to" attribute containing the JID of the WordTask that initiated the WorkFlow. TheWordWorker, after having completed its work, forwards the WorkPacket to the indicated JID for further processing.
Note that it is possible to have a WorkFlowStep contain a non-null “toRoom” attribute, which would cause the forwarding of the WorkPacket to a room (for further processing, for example). This feature is not tested or demonstrated yet, although it is available in the code.

Finally, if a WorkFlowStep has no “toRoom” or “to”, the assumption is that the WordWorker should continue to process the next WorkFlowStep, if any. This allows the work to stay at one location and perform multiple steps in sequence. This feature is not demonstrated by the WordTask demo, but it is potentially available in JabberGrid.

### 2.4.8 WordTask receives WorkPacket back from WordWorker

Note that when WordTask receives this response, it is via point-to-point messaging (as contrasted with room-based multicast). When the WorkPacket is received, its RoutingSlip has been adjust so that it points to “Step2.Collect”, which the WordTask interprets and collects the transformed text from the WorkPacket payload and stores it in the WordTask object for later display.

### 2.4.9 WordTask detects task completion

The final step in a Task is for its isTaskComplete() method to return true, which usually unblocks the task from its waiting state and lets control return to the creator of the task. In the case of WordTask, isTaskComplete() will return true when the Task has received as many Step2.Collect WorkPackets as it issued initially.

At this point, the main program detects the completion and prints out the transformed texts.

### 3. Examples of Distributed Work
There are a wide variety of potential applications for distributed problem solving (DPS) techniques, including: sensor interpretation in distributed sensor networks; vehicle traffic monitoring and control; detection and diagnosis of faults in computer and telecommunications networks; information retrieval from distributed information sources; computer-supported cooperative work, and various kinds of situation assessment [2]. DPS approaches can potentially offer increased processing power, flexibility, and reliability, and reduced hardware costs.

Although there are a variety of interesting problems in computer science that are amenable to distributed work, we chose 'simple' problems that emphasize the issues relating to our choice of communications medium, rather than emphasizing the knottiness of the problem itself.

The tasks we built for JabberGrid let us explore many of the essential problems in distributed work systems. While the tasks themselves are trivial, the process of solving them is not.

The JabberGrid project focused on the following tasks.

3.1 The 'WordWorker' Task

This problem can be characterized as a text processing task. The problem is for the system to make transformations on a large packet of data. We choose the transformation to be to uppercase the data, but any kind of text processing problem could be solved in a similar manner.

A Task creates a room and injects the problem (uppercase data). The data is broken up into smaller packets and sent to the room. Workers in the room compete for the work packets until all the parcels of data are claimed. The workers compute the transformation and send the altered data back to the Task. Workers can perform work
while the Task is still injecting the data to increase performance

In the JabberGrid system, communication can be multi-cast or point-to-point. When the work is injected into the room, a message is multi-cast to all the workers listening to that room, but when the workers want to send back the completed data it communicates directly to the Task that hired it. This functionality helps reduce the bottleneck of having to collect and send the large data packet back through one channel.

WordWorker is the core task implemented on JabberGrid. It entails many important problems in distributed environments and systems such as; action coordination, task scheduling and resource allocation (see Section 4).

3.2 The 'PingPong' Task

In this task the work is completed in a workflow of multiple (or the same) workers doing some work and then passing the data to the next worker it is assigned to. This task models the manila envelope metaphor used in offices.

A Task injects an 'envelope' that contains the problem, the data, and the next worker to send the packet to. The first worker does its job on the data, replaces the data in the 'envelope', crosses its name off and forwards it to the next worker on the address.

This Task explores the workflow capabilities of the project. In JabberGrid a Task knows the names and services of the workers in its room. Implementing the text processing task to include different transformations (eg language translation) or multiple transformation is a trivial extension of this Task.

3.3 The 'WordWorkerPool' Task.

This task explores the idea of a hiring pool. Instead of a Task injecting work into a room
full of workers it hires workers through a manager of a 'WorkerPool' of workers. The idea is that a WorkerPool would have a diverse set of workers with different "skills" (service levels). By sending a "hire" request (a resource acquisition, really) with appropriate desired "skills" indicated (service levels, in the grid nomenclature), the appropriate Workers can be selected. This is sometimes refered as a "Trader" service (eg CORBA).

In effect, WorkerPool rooms are like matchmaking services that match up employers (tasks) with specific needs with employees (workers). Once a match has occurred, however, the dictates of the problem require that the Worker "move" to a task-specific room to perform their work. In this Task, the Worker moves to a specific room that contains a specific set of Work. Thus, one could theoretically have several different WordTasks, each of which had its own room and set of words. These tasks could share the same WorkerPool, hiring and releasing workers from it.

The indirection provided by moving the workers from the hiring room to the work room offers the flexibility of not instantiating a worker in a hardcoded room (as in the WordWorker Task).

These 'simple' tasks demonstrate the potential of JabberGrid. They support our conclusion that JabberGrid is capable of solving many non-trivial problems and we believe further investigation in JabberGrid could be worthwhile (see Section 6).

Problem suitability is further discussed in Section 4.5.

4.Challenges and Areas of Investigation

Distributed problem solving systems are concerned with how large-scale problems can be solved using systems of agents, distributed among a set of networked computers, working cooperatively.
The basic concept of problem solving systems is problem decomposition; the ability to take a large, hard problem and break it into many smaller problems. Then, give the smaller problems to a computer or client that has nothing else to do and let them do the work.

There are many questions that must be addressed before this simple transaction can be carried out, such as:

- How does a system discover workers and how do the workers discover work?
- How are the problems distributed so that redundant work is not performed?
- How do you collect and collate results such that bandwidth does not become a problem?
- How can workflow be utilized to increase the scalability of a system?
- What guarantee exists that the work will be completed and returned?

To satisfy these questions a system must offer the following functionalities.

- A look-up service.
- The ability to discover services.
- Coordination of workers and resources in order to reduce redundancy and improve performance.
- Efficient use of network bandwidth and good performance.
- Security
- Fault tolerance.

4.1 Challenge: Worker Discovery and Coordination

A federation of services, is a set of services, currently available on the grid, that a client (meaning a program, service, or user) can bring together to help it accomplish
some goal. The idea behind the word *federation* is based on the idea that in a grid there isn't a central controlling authority. Because no one service is in charge, the set of all services available on the network form a federation, a group composed of equal peers. JabberGrid used the federation idea to decompose the problem.

One of the ideas of the JabberGrid project is that the distributed system works by having the workers choose their subtask of the work package to work on, rather than having a central coordinator define which worker gets which task. Using the federation idea the system can adaptively configure itself to varying loads and different worker capabilities.

To enable a separated group of agents to collaborate, they need to coordinate themselves. Examples of the need for coordination in collaborative work is the need to ensure the completion of all work, the lack of redundant work (e.g. avoid conflicting actions) and the timely completion of the work.

The cooperation of agents which engage in a common task requires the coordination of the task-related activities as well as the coordination of the resources used during the execution of these activities.

JabberGrid uses a SmartMulticast that guarantees that a message issued to multiple users is only delivered by one. In effect, the possible recipients of a message use a 'consensus' algorithm to select a chosen recipient.

### 4.1.1 Consensus in JabberGrid

One of the most important performance problems in a distributed system such as JabberGrid is:

- How do you make sure that the work is carried out only once?
- Or, how do you make sure that not every worker carries out the same work?

*Example Solution:*

Use a three way handshaking algorithm, where every worker who wants to carry out a
particular work (in a room) handshakes with the owner (injector of the work). This way the owner ensures that only one person carries out the work. Both the problems are solved.

_Problems with the Example Solution:_

- If there are ‘n’ workers in the room, then for every work packet, the owner (injector of the work) will receive ‘n’ requests in the first round of handshake. (If point to point messaging is used).
- The owner (injector of the work) may crash if the value of ‘n’ is too big.

### 4.1.2 SmartMulticast in JabberGrid

SmartMulticast is a distributed approach to solve the consensus problem. When a worker sees a work-packet, it waits for a random amount of time. (Random Back off mechanism). After the wait time has elapsed, the worker checks to see if the work is still available. If it is, it claims the work, else it knows someone else has. To claim the work, the worker has to send an acknowledgment (for that particular work) to the room. To know if the work is already claimed, the worker has to check to see if there is ‘one’ acknowledgment (for that particular work) in the room.

**Advantages of this approach:**

- Since every worker randomly backs-off, only the worker with the smallest back-off time will claim the work. All other workers will see the acknowledgment of the first worker. Both the performance problems are solved.
- Since the workers resolve the problem amongst themselves it is transparent to the owner (injector of the work). i.e. there is a clear seam in terms of functionality, where the owner provides the work and workers carry it out. The injector does not have to bother about how the work is carried out

### 4.1.3 SmartMulticast Implementation Details
In JabberGrid, messages are sent/received asynchronously. This means that as soon as a work packet is received (by the room) it is delivered to the workers. Now, the workers may be in one of the following states:

- State I – Idle and waiting for work
- State II – Carrying out some work
- State III – Trying to claim work

If the worker in State I is interrupted with work, then its fine, but workers in State II and State III would not want to be interrupted.

To ensure that the system does not interrupt any workers, we have used a queuing mechanism, where the incoming messages are stored in queues. Now, if the worker is in State II or State III, he will not be interrupted with any new messages, as all incoming messages will be queued (exception to this is the system messages). Also, any worker in State I can probe his queue to see if any new request has arrived.

**4.1.4 SmartMulticast Algorithm:**

For each worker:
- Each incoming work/hire packet goes into the mainQueue.
- Each incoming acknowledgment goes in the ackQueue.
- If mainQueue is not empty then remove the first packet.
- Wait for a random amount of time.
- Check to see if the ackQueue contains any acknowledgment corresponding to the packet removed.
- If none, send an acknowledgment claiming the work.
- Else go to 3.
- Detect Collisions, If none carry out the work
• Else resolve conflicts by comparing the counter value.
• If counter value is the greatest carry out the work, else leave it.
• Go to 3

4.2 Challenge: Service Discovery and Bootstrap

In distributed systems, a discovery service that provides lookup based on service properties is generally referred to as a trader. A consumer first decides what properties it desires from the service it’s seeking. The trading service matches those properties against the properties of the services registered.

Given that service discovery depends on discovery services, how does an application actually find a discovery service? Relying on static configuration data to bootstrap an application with respect to discovery services is the approach our initial prototype has taken. A future direction is to use the XMPP presence mechanism.

Workers need to discover their Jabber server and the appropriate Jabber channel to communicate on. Currently this is hardcoded into the workers, but could easily be parameterized or looked-up via DNS.

Lookup services are organizing mechanism for distributed systems. When new services become available on the grid, they register themselves with a lookup service. When clients wish to locate a service to assist with some task, they consult a lookup service.

JabberGrid does not currently support a lookup service. But, the project could easily use the presence mechanism (see section 2) in the Jabber technology as the means to
enable smooth adding, removal, and finding of devices and services on the network. Presence lets users of the system know who is on the grid and what services are available. This information could be queried by a Task looking for workers and enables a Task to register its room for workers to discover.

In the best known distributed problem solving system, seti@home, there is a designated place that clients know to go to find work. JabberGrid is not much different in that individual machines looking to offer services can log into known Jabber rooms on the Internet.

4.3 Challenge: Workflow

Workflow applications are distributed by nature and often are aiming at goals such as reliability, scalability, and efficient load distribution in problem solving.

Workflow demonstrates the area of process interoperability: the linking of processes to carry out complex computational tasks. By utilizing workflow, JabberGrid can accelerate throughput of a problem.

In JabberGrid workflow is implemented with routers and headers (see Section 2). The Task uses a workers unique Jabber ID as the routing information in the header. When a worker receives the work packet it acts as a worker (does some work) and a Task (forwards work to the next worker in the header).

Workflow management can be coordinated through the point-to-point communication or the smart multicast available through Jabber. Using point-to-point communication, workers can send the packet through their own channels without sending the data back through the room. While using multi-cast communication allows work to be easily distributed amongst workers.
4.4 Investigation: Scalability, Security and Fault Tolerance

One of the obvious metrics to use on the JabberGrid project are performance issues such as bandwidth, number of messages used to solve problems, and fault tolerance.

One attraction of the Jabber technology is the possibility that multiple servers and workers can be organized in topologies such that no single server needs to handle the communication for all of the workers. This provides some fault tolerance in the project as well as avoiding performance bottlenecks.

In JabberGrid the problem occurs when a worker claims a work packet and then crashes. If the worker does not restart the work packet is lost. But, the worker can come back because of persistence in Jabber technology and reclaim work since it can see that it had previously claimed it. Most distributed work systems have the problem of guaranteed returns. In the seti@home project this problem is combated by sending out duplicate work packets. This increases the probability that work will be completed and returned but introduces potentially unnecessary redundancy into the system.

Another advantage that Jabber technology provides is security measures. All Jabber servers are isolated from the public Jabber network, and robust security using SASL and TLS has been built into the core XMPP specifications [2]. For server-to-server communication the initiating JID is used for authorization. For client-to-server communication the bare JID (user@domain) is used as the authorization identity and client-to-client communication is initiated through the client-to-server authentication as well.

On the server side JabberGrid has almost unlimited scalability because any Jabber server can talk to any other Jabber server that is accessible via the Internet. But, because Jabber technologies use a client-server architecture, JabberGrid suffers from a client bandwidth bottleneck when passing large work packets back to one Task. Evaluations showed that the bottleneck occurs for over 2.2 megabytes of data.
4.5 Investigation: Problem Suitability and Integration

Certain types of work are more amenable to distributed problem solving than others. The particular characteristics of the Jabber network will favor problems where the worker needs to spend moderate to large amounts of time to generate a result. In addition, there is a relationship between the payload consumed/generated by the worker and the amount of time that the worker needs to solve the subtask.

4.5.1 Recursive Problems

JabberGrid can be used to solve certain types of recursive problems. To solve recursive problems the different workers that process the work must synchronize with their children so that there is a tree of workers all waiting for the leaf nodes to complete.

Although we did not implement any recursive algorithms we considered them and have added support for them in the project.

Consider the workflow system we implemented as a distributed programming language with each workflow step as an instruction, then extending this to solve recursive problems is easy to imagine. For example, suppose that we created a MergeSortWorker, which received a set of elements to sort. Suppose this worker created new WorkPackets and sent them to a WorkerPool or specific MergeSortWorker to work on, while it waited for them to complete. When they complete and return their results, the parent MergeSortWorker merges the results and sends the response to its parent. The MergeSortWorker is acting both as a worker AND as a Task (because it waits for responses from its subworkers).

4.5.2 Monte Carlo Simulation
Monte Carlo Simulation are another class of suitable problems for JabberGrid. Monte Carlo simulations randomly generate values for uncertain variables over and over to simulate a model. The simulation calculates multiple scenarios of a model by repeatedly sampling values from the probability distributions for the uncertain variables.

In contrast to recursion, in Monte Carlo simulations the Task is completed by distributing the work of simulating random variables to multiple workers. Each worker can do its work without synchronizing with any other worker. Each worker just needs to simulate a particular number of random variables and return an average "score" to the Task.

In Grid research these types of problems are often called "embarrassingly parallel", because they are so easy to decompose.

The Grid architecture enables the Integration of services and resources across distributed organizations. Such a platform provides capabilities ranging from integrated resource management, clustering services, workload management, and problem solving.

Jabbergrid has the potential of providing these capabilites. Through the power of XML namespaces, custom functionality can be built on top of the core protocols. This means the Jabber technology is very extensible. Therefore, JabberGrid can support many types of integration with any type of distributed systems. Some ideas we discussed were; RMI/SOAP, Googlebot agents, and webcrawlers.

5.Evaluation Results

To evaluate the performance of JabberGrid we compared it to Java RMI. The speed we were interested in was not raw throughput, but how long it took to do the work. To track the results we used a set of benchmarks that includes an empty payload, a big payload(1 megabytes), and a very large payload (2 megabytes).
The graph below shows the speed of JabberGrid compared to JavaRMI.

Notice that JabberGrid performs as well as RMI up to one megabyte of data. After this size of packet RMI becomes almost twice as fast as JabberGrid. After two megabytes of data, Jabber Grid experiences a bandwidth bottleneck. JabberGrid is implemented over XML which leads to a considerable amount of overhead. Nevertheless, the performance of JabberGrid when compared to RMI is impressive for work packets under one megabyte.

We evaluated the scalability of JabberGrid with two metrics: throughput and packet size. We were interested in how large of work packets could JabberGrid solve before experiencing a bottleneck in bandwidth. To track the results we used a set of benchmarks consisting of an empty payload, 1 megabyte payload, 2 megabytes, etc.

The graph below shows the packet size and throughput time. You can see that up to one megabyte, throughput time is adequate. The real scalability bottleneck happens for over two megabytes. Some of the time it takes to solve a problem is spent waiting for a worker to claim the work packet in the SmartMulticast. With more time we could improve the consensus algorithm to allow for better time. Evaluation seem to show two megabytes of data is the maximum size of work packet JabberGrid can successfully
Figure 4. Throughput and Packet Size Scalability

6. Future Work

These are some outstanding issues and new implementations we envision for the future of the JabberGrid project:

- Enhance work flow packets.
- Implement presence to use as a discovery services.
- Test Server-to-Server Communication.
- Build more interesting workers – translators, stock ticker, etc..
- Build a distributed game using JabberGrid as communication architecture.
- Compare GRID specification in OGS to JabberGrid.

7. Conclusion

The goal of the project was to show that Jabber technology has the potential of being an important keystone technology in grid computing. JabberGrid demonstrates this potential by using Jabber to implement a distributed problem-solving system.

JabberGrid uses the idea of a Grid as as a sort of marketplace that brings users (service consumers) and worker (service providers) together for the purpose of solving a common task. The task can be computationally unmanageable for one agent to compute alone or even if an agent could in principle solve the large task alone, solving subtasks in parallel can yield an overall solution faster.
JabberGrid was used to solve some 'simple' tasks that revealed technical issues that are common to any distributed problem solving effort. The project solved many of those issues such as; worker and service discovery, distributed coordination through a consensus algorithm, and workflow management. The project explored the possibilities inherent in the Jabber technology such as scalability and fault tolerance mechanisms.

The simple task we implemented demonstrate that JabberGrid can be used for solving non-trivial problems like those described in the problem suitability section (4.5). The Jabber network favors problems where the worker needs to spend moderate to large amounts of time to generate a result.

The use of JabberGrid to solve problems must offer some compensation in payload consumed/generated by the worker and the amount of time that the worker needs to solve the subtask. Evaluations of JabberGrid showed that it can handle an impressive size of workpackets (2.2 megabytes) before it has bandwidth issues. Evaluations on performance show that JabberGrid performs as well as Java RMI for considerable large size of data.

Further work is needed to make JabberGrid a real grid computing system, but it has been shown that it is possible to use Jabber for such a purpose.

References


[4] Study on Developing a Distributed Problem-Solving System. Eun Gyung Kim
Appendix A – User Documentation

It is assumed that the user has all required Jabber technologies and Ant Apache installed and a Jive messenger server started on their machines.

• **Build Directions:**

  1. Type `ant` into the command line of the directory where JabberGrid was installed.
  2. The user should see a message that says “Build Successful”

• **Run Tasks**

  • **WordWorker Task**
    1. Open two terminals
    2. Type `ant wordtask` into the command line of one terminal.
       - This create a wordtask room, poses a problem, and signs up
workers.

– The user should see messages indicating that the Task created a room and injected the data to be transformed.

3. Type ant wordworkers into the command line of the other terminal.

– This starts up some workers in wordtask room

– The user should see messages indicating that workers were created and that they joined the room, took some work and completed the task.

– Simultaneously the wordtask terminal will display messages of the handshake mechanism that happens when a wordworker joins a room.

– The results will be displayed once the problem is completed and returned to the word task.

• Ping-Pong Task

1. Open two terminals

2. Type ant pingpongtask

3. Type ant pingpongworker

• WordWorkerPool Task

1. Open three terminals

2. Type ant workerpool into the command line of the first terminal

– This creates an empty workerpool room.

– The user should see a message that the room was created.

3. Type ant wordworkerspool into the command line of the second terminal.

– This creates a worker factory.

– The user should see messages that workers have been created have joined the workerpool room.

– Messages should be displayed in terminal one displaying the handshake mechanism that happens when a wordworker joins a room.

4. Type ant wordtaskpool into the command line of the third terminal.
- This creates a work room and hires workers from the workerpool room.
- The user should see messages that the room was created, and the hiring and firing task instructions. When the work is completed the results of the problem are displayed.
- In terminal 2 the user should see messages that the workers were hired and moved to the work room.
- In terminal 3 the user will see the handshake mechanism that occurs when a worker disconnects, then reconnects to the workerpool room.