Solving Quarto with JXTA and JNGI

Matthew Shepherd
University of Texas at Austin

Abstract. This paper presents an implementation of a grid-based autonomous quarto player. The implementation uses the JNGI framework which itself is written on top of JXTA.

1 Introduction

Quarto is a game reminiscent of tic-tac-toe. It is played with sixteen unique pieces on a four-by-four square board. Each piece is large or small, black or white, solid or hollow and square or round. The game begins with an empty board and all sixteen pieces available. Two players take successive turns with the first player choosing an available piece. That piece is given to the second player who places it on an empty spot on the board. The roles reverse and the process repeats. A player wins by placing the final piece in a row, column or diagonal where all four pieces have at least one attribute in common. For example, the winner might place a piece that completes a row of four black pieces or a column of four round pieces.

My goal was to write an implementation of the game and an autonomous program for a human player to compete against. The program performs an exhaustive search of possible moves in order to find the most appropriate one. If the program were to perform the search on the first move of the game, the sixteen available pieces and sixteen empty squares would combine to produce a (16!)^2 search space. Eliminating the moves that take place after a game has already been won reduces that number, but a single processor would still not be sufficient to perform such a task in a reasonable amount of time. I proposed implementing the program to run on a grid of available computers hoping to reduce the running time to a more desirable time span, allowing a user to play against the program in real-time.

2 Approach

The purpose of the project was to supplement my exposure to grid-based systems in the classroom with firsthand experience. I felt that the scope of that experience needed to be restricted to working with a single existing framework in order to gain a sufficient depth of understanding. I chose to use JXTA as the underlying framework for the game based on my strong familiarity with Java and related technologies from Sun Microsystems.

2.1 JXTA

JXTA is a response to the growing number of isolated peer-to-peer applications. These networks often have two flaws in common. First, the advent of each new peer-to-peer application usually means the introduction of another set of peer-to-peer communication protocols. These protocols are the way that peers discover and communicate with each other. The same base set of communication protocols are needed by virtually every peer-to-peer application. By implementing the protocols needed to support the new application, the application's developer frequently writes software that has been written many times before. Had the developer used an existing framework, the time spent implementing the underlying protocols could have been put to better use. Second, the application does not take advantage of its peers acting as peers another network. Many operations, such as discovery of other peers, might be consolidated across several applications running on an individual host thereby freeing up system resources for other uses.

With these goals in mind, Sun Microsystems publicly announced Project JXTA in February 2001. With the help of researchers at academic institutions, Sun had defined a set of peer-to-peer communication protocols independent of any platform or programming language. The protocols define a set of XML documents that peers can send and receive. These documents can be used to transmit requests or responses among peers to provide a base set of functionality such as discovery and message routing. These documents can also be extended for application-specific purposes like coordinating workers and distributing code in a grid-computing application. The JXTA specification also introduces the concept of peer groups. It suggests that peer groups can be used to, among other things, define the scope of messages and administer security.

Sun released the JXTA reference implementation as open source through jxta.org in April 2001. The initial reference implementation, written in Java, consists of a set of services that give life to the JXTA protocols. Developers writing peer-to-peer applications can do so on top of these services at no charge. Like other open source projects, it also encouraged developers to contribute bug fixes and improvements back to jxta.org. These modifications along with the work done by the core team eventually lead to the release of the version used in the implementation of quarto: JXTA v2.1.1.

2.2 JNGI

Many developers have sought to build extensions to JXTA. The extensions enable others to more easily write complex peer-to-peer applications. One such project is JNGI. JNGI was first proposed in 2002 in a paper titled "Framework for Peer-to-Peer Distribution Computing in a Heterogeneous Decentralized Environment" by Jerome Verbeke, Neelekanth Nadgir, Greg Ruetsch and Ilya Sharapov at Sun Microsystems. The paper described the framework for a grid of JXTA peers that could be used for various computationally intensive problems. The primary goal of JNGI is to enable a developer with little knowledge of JXTA to write a program that taps the computational power of available JXTA peers.

JNGI defines a Monitor, Worker and Task Dispatcher peer group. Membership in these peer groups is used to define the roles of the participating peers. A peer can be a member of one or many groups and there can be one or many instances of each peer group in the JNGI network. A request to join the JNGI network is handled by the Monitor peer group. The monitor group also determines which groups and roles to assign to that peer.

The Monitor peer group grants a job submitter membership to the network, like any other peer. Once granted membership, the job submitter submits a new job consisting of logic and data. The logic takes the form of a Java class, implementing java.lang.Runnable and java.lang.Serializable, compiled into standard byte code. All worker peers involved in the same job will run the same Java class. Where their work differs is in the data. The data takes the form of serialized instances of the Java class. JNGI refers to these instances as tasks. These tasks are instantiated with different data before they are serialized and distributed.

The task dispatcher group receives the byte code and tasks from the job submitter. The worker peers have been polling the task dispatcher requesting tasks since they were granted membership to the JNGI network. Now that the task dispatcher has a set of tasks to distribute, it responds to the workers with the byte code, if necessary, and a task. The worker uses the byte code to de-serialize the task and then runs the object as a normal Java Thread. The object completes its task and sets the result as a member variable. It is then sent asynchronously back to the Task Dispatcher group. It is possible for the Task Dispatcher group to send out the same task to multiple workers. In this case, the Task Dispatcher uses the result returned first and discards the others.

Since submitting the job, the job submitter has been polling the task dispatcher requesting the completed tasks. Once the task dispatcher receives a response to each task, it in turn responds to the job submitter with all the serialized objects that were returned by the workers. The job submitter is responsible for any post processing that might be necessary.

2.3 Quarto Implementation

The quarto implementation uses JNGI v1.0 and JXTA v2.1.1, the most recent stable releases at the time of this paper. The quarto grid uses a set of hosts on a single isolated Local Area Network (LAN). Each peer configures JXTA to use TCP on a single port with multicast enabled and HTTP access disabled.

2.3.1 Working with JNGI

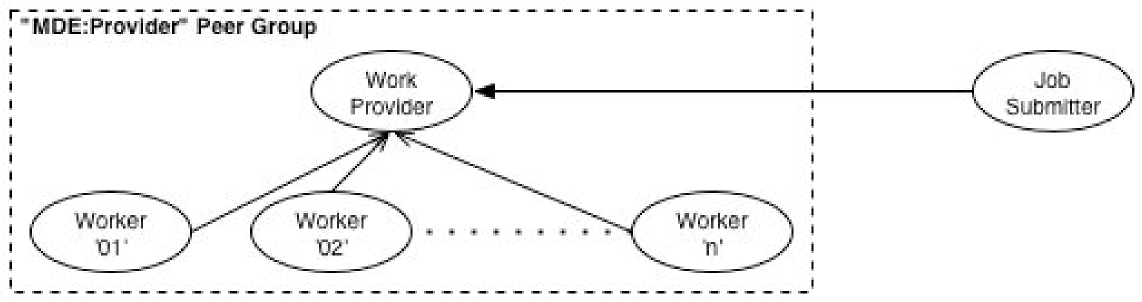

Figure 2.3.1

The JNGI implementation is a much-simplified version of the framework described in "Framework for Peer-to-Peer Distribution Computing in a Heterogeneous Decentralized Environment." The implementation uses only a single peer group, MDE:Provider, that is instantiated by the work provider peer. In the JNGI implementation, the worker provider takes the place of the Task Dispatcher group described in the JNGI paper. Once the work provider is running and the MDE:Provider peer group is available, the worker peers can be started. The work provider and worker peers should both be running prior to the start of the quarto game.

The quarto game interacts with the JNGI network using a RemoteThread. The RemoteThread class mimics the java.lang.Thread API familiar to Java developers. This abstracts away the job submitter's interaction with the work provider thus making the source code using it fairly straightforward.

2.3.2 Quarto Classes

quarto.Game. The Game class implements the *main* method and can therefore be invoked from the command line by the user. It instantiates two Player objects and one Board object, using them to control the flow of the game.

quarto.Board. The Board class maintains the state of the game including which marks (pieces) have been played on which spots (squares).

quarto.Player. The Player interface defines the getName(), chooseMark() and chooseSpot() methods that allows the Game object to treat all player implementations uniformly.

quarto.User. The User class is the Player implementation that provides the command line interface that allows a human player to choose marks and spots as the Game object requests them.

quarto.Robot. Deprecated. The Robot class is an autonomous serial implementation of the Player interface. It uses the min-max algorithm to choose marks and spots as requested by the Game object. The class was deprecated in favor of the GridRobot implementation.

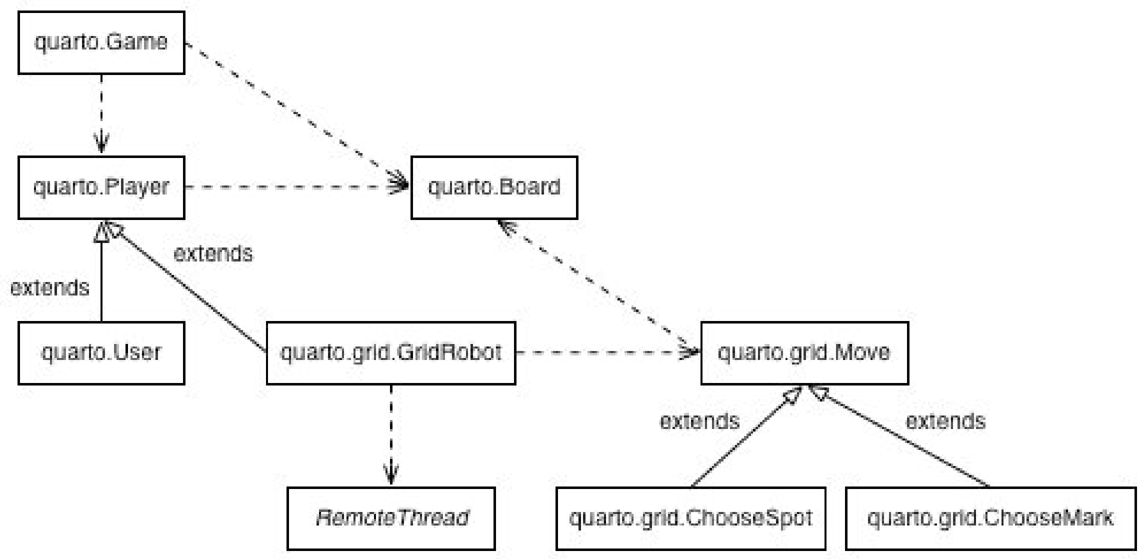

Figure 2.3.2

quarto.grid.GridRobot. The GridRobot class is an autonomous grid implementation of the Player interface. It uses the min-max algorithm in concert with the JNGI RemoteThread class to choose marks and spots. The GridRobot is responsible for dividing up the search into JNGI tasks and making sense of the results.

quarto.grid.Move. The Move class is an abstract class that implements Serializable and Runnable. This allows its subclasses to be distributed as JNGI Tasks.

quarto.grid.ChooseMark and quarto.grid.ChooseSpot. Theses classes extend the Move class thereby implementing Serializable and Runnable allowing them to be distributed as JNGI Tasks. The two classes perform an exhaustive search of the possible boards and then choose the best available mark or spot.

2.3.3 Quarto in Action

The quarto implementation consists of two java packages. The first package, *quarto*, contains the necessary classes to play a command line game between two human players situated at the same computer. The second package, quarto.grid, provides the implementation of the grid-based autonomous quarto player.

When the quarto game begins, a Game object is instantiated. The Game object in turn instantiates two Player objects and one Board object. At least one of the Player objects will be a GridRobot. The GridRobot acts as the job submitter. The Game will ask the GridRobot to choose a mark or a spot depending on its turn. The GridRobot will then instantiate an array of Move objects. The array will consist of either one ChooseSpot object for each available spot or one ChooseMark object for each available mark. This array of Move objects, along with the Move, ChooseMove, ChooseSpot and Board classes, are sent to the work provider using a RemoteThread. Control is given back to the GridRobot once all Moves have completed their search. The GridRobot sorts through the completed Moves, chooses the first best Move and returns it to the Game.

The process repeats itself on each of the GridRobot's turns. Unfortunately, sending a second job within the same process is a limitation of the existing JNGI implementation. JNGI attempts to "boot" a new peer each time a RemoteThread is started, but at this point the peer is already running. The result is that JNGI crashes and the quarto game ends prematurely. I was able to modify the RemoteThread class to hold a reference to the existing peer so that it can be reused. Using the modified JNGI code, the GridRobot was able to play more than a single round of quarto.

3 Results

When a worker receives a task, it must either complete that task itself or not complete it at all. JNGI does not support a worker dividing up a task further and redistributing its portions to more workers. This limitation presents an insurmountable hurdle for the quarto implementation. A single level of distribution can only reduce the initial (16!)^2 by a factor of 16. The search space remains too large for a single host to process. To verify the implementation worked correctly, the game was seeded with several predetermined moves. These limited tests proved successful, but given the current implementation of JNGI it is not practical to play the game from the very beginning.

4 Conclusions

Aside from the lack of branching jobs, JNGI has other serious limitations that were encountered during the implementation of quarto such as the platform dependent shell scripts. The scripts are intended to ease the deployment of the peers, but they are both platform and network dependent making them useless in a typical environment. The problem is compounded by the lack of documentation regarding peer configuration and the lack of community involvement on the project website. This results in a long painful attempt to simply run the example application provided.

Once the sample application does run, however, it becomes clear that the JNGI implementation differs from the paper that introduced it in several key areas. The three peer groups described in the paper (Monitor, Worker and Task Dispatcher) are not found in the implementation. Replacing the Task Dispatcher group is a singe peer known as the work provider. The instances of Worker groups seem to have been replaced by a lone

MDE:Provider group and the Monitor group is missing entirely. The three peer groups are the heart of the JNGI framework described in the paper. By removing them, the implementation has, at best, a vague resemblance to the paper.

The simplification of the JNGI paper and the inability of workers to invoke more workers cripples the framework. It is necessary for a grid-computing framework to offer at least some rudimentary form of task branching in order for it to successfully solve difficult computational problems. The JNGI framework has a solid foundation in JXTA because the JXTA peer groups offer a great deal of flexibility in building a grid-computing framework. Unfortunately, that flexibility is not presently passed on to developers using JNGI and unless it is, the JNGI framework and its implementation will be of little use to them.

5 References

1. Jerome Verbeke, Neelekanth Nadgir, Greg Ruetsch, Ilya Sharapov. Framework for Peer-to-Peer Distribution Computing in a Heterogeneous, Decentralized Environment, 2002, Sun Microsystems, Inc., Palo Alto, California.
2. Jerome Verbeke, Neelekanth Nadgir. JNGI: P2P Distributed Computing, September 2003, Sun Microsystems, Inc., Palo Alto, California.
3. Project JXTA Technology: Creating Connected Communities, March 2003, Sun Microsystems, Inc., Palo Alto, California.
4. Li Gong. Project JXTA: A Technology Overview, November 2002, Sun Microsystems, Inc., Palo Alta, California.
5. Bernard Traversat, Ahkil Arora, Mohamed Abdelaziz, Mike Duigou, Carl Haywood, Jean-Christophe Hugly, Eric Pouyoul, Bill Yeager. Project JXTA 2.0 Super-Peer Virtual Network, May 2003, Sun Microsystems, Inc., Palo Alta, California.
6. Project JXTA v2.0: Java ^{TM} Programmer's Guide, May 2003, Sun Microsystems, Inc., Palo Alta, California.
7. J. C. Browne, K. Kane, H. Tian. An Associative Broadcast Based Coordination Model for Distributed Processses, 2002, The University of Texas, Austin, Texas.