IEEE Transactions on Software

Design and Architecture of Jxta-Based Game Software in Java.

Maia Software.

Frank Appiah

University of London, King's College London, Strand, London, WC2R 2LS, United Kingdom

ABSTRACT. *This paper discusses the interest of explicit software processes of design and architecture for (1) architectural view understanding and communication, (2) design reuse and principles of Jxta-based game software development. The considered architecture includes hierarchical and logical views which are modeled in the Java-based Maia API and achieved by Jxta message exchanges. The topology of the world map of the game software can be viewed as undirected graph in which vertices (nodes) represent the rooms and the edges represent the playable movements between nodes.*

Categories and Subject Descriptors

H.4 [Software]:

; D.2.8 [Software]: API development—Java

General Terms

Design, Graph Theory

Keywords

XML; P2P; XML Parsing; Software; Game

1. INTRODUCTION

In order to share resources, peers in Maia interact directly with each through the exchange of messages. Peers directly access and exchange the share resources they utilize in without a centralized service. A framework called shoal¹ based on the technology, JXTA [8] used in the design of the peer-to-peer game software has the notion that for cross subnet communication like the internet there needs to exist a virtual list of hosts which will be used to route all messages to other peers over the internet. Peer-to-Peer (P2P) software is simply a software system with a completely decentralized self-organization and resource usage. P2P mechanisms can be used to access any kind of distributed resources and may offer any new internet-based applications mostly in the area of gaming [14]. In order to achieve that we introduce a peer called Server which is logically another peer with the same shared resource as the other peers, Maia Player-Peer. This model offers adequate formal basis to deal with the reliable design of distributed software. It has become important that for internet-based applications, issues of scalability, performance, security, flexibility and quality of service

¹ Shoal is a java based scalable dynamic clustering framework that provides infrastructure to build fault tolerance, reliability and availability. The framework can be plugged into any product needing clustering and related distributed systems capabilities without tightly binding to a specific communications infrastructure. <https://shoal.dev.java.net/>.

needs to be taken care of. These requirements are some good benefits which P2P offers in the main architecture of P2P software. The Server is a Web Server precisely Tomcat [16] and this will enable the application to continue working once Tomcat is up and running. Graph [1] played an important role in defining the notion of the world map.

The rest of paper is organized as follows: In Section 2, the overall overview of Maiar architectural views with detail explanation of its 5 components. An introductory material and the design principles of each of the major 5 components are provided. Section 3 describes the game component model. In Section 4, I will look at the design principles of the messaging component model. In Section 5, the principles of design of the game component model are described. Again, Section 6 describes the principles of the logging component. Section 7 is the look at the security component model in details. Section 8 will be the architecture of Maiar that will be accessed on how it provides some major software quality attributes. In Section 9, I look at the deployment view of Maiar game software. Finally, I conclude this paper.

The main contributions achieved in this paper are as follows:

- *The architectural design of peer-to-peer game software in Java.*
- *The overall design of the main component models of Maiar Architecture.*
- *The design principles of a game component model.*
- *The security concerns of an internet game.*
- *The overall action and effect life cycle of Maiar game.*

Figure 1 Hierarchical View of Maiar Architecture

2. ARCHITECTURAL VIEWS OF MAIAR

In this section, we will start with the representation of world and discussion of the major components of the Maiar API with concise reasons of choice of technology. The API is comprised of a number of components, specialized for a specific purpose. Layers and domain objects are applied to the design of Angels and Demons game called Maiar to introduce a clear identification and modular separation of all baseline architecture elements in both their

responsibilities and core usage relationships. As such vertical and horizontal decomposition of the game's functionality is realized using Apache Maven [10]. **Figure 1 depicts the** hierarchical view of Maiar Architecture and Figure 2 depicts the logical layer of the Maiar Architecture.

Figure 2 Logical Layers of Maiar Architecture

2.1 World Representation

In this section, I will have a look at the algorithm for the XMLtoObject parser and the structure of the XML [3], [15] document.

2.1.1 XML Structure of World.

The xml structure of the world is defined with a DTD schema. The elements used in the representation of the World are as follows:

```

<!ELEMENT map (exit / global / rooms /
  entry)*>
<!-- map
  name CDATA #IMPLIED
  <!-- rooms (room)*
  <!-- room (commands/directions/
    items
    / look / intro)*
  <!-- room

```

```

  id CDATA #IMPLIED
  name CDATA #IMPLIED
  <!-- intro EMPTY
  <!-- intro
  message CDATA #IMPLIED
  <!-- look EMPTY
  <!-- look
  description CDATA #IMPLIED
  <!-- items (item)*

```

```

<!ELEMENT item EMPTY>
<!--ATTLIST item
name CDATA #IMPLIED>

<!--ELEMENT directions
(north/south/west/east)*>
<!--ELEMENT east EMPTY>
<!--ATTLIST east
roomId CDATA #IMPLIED>
<!--ELEMENT west EMPTY>
<!--ATTLIST west
roomId CDATA #IMPLIED>
<!--ELEMENT south EMPTY>
<!--ATTLIST south
roomId CDATA #IMPLIED>
<!--ELEMENT north EMPTY>
<!--ATTLIST north
roomId CDATA #IMPLIED>
<!--ELEMENT commands (command)*>
<!--ELEMENT command (action)*>
<!--ATTLIST command

```

```

name CDATA #IMPLIED>
<!--ELEMENT action
(requirement/effect)*>
<!--ELEMENT effect EMPTY>
<!--ATTLIST effect
operator CDATA #IMPLIED
value CDATA #IMPLIED
parameter CDATA #IMPLIED>
<!--ELEMENT requirement
(notSatisfied/satisfied)*>
<!--ATTLIST requirement
value CDATA #IMPLIED
parameter CDATA #IMPLIED>
<!--ELEMENT satisfied (action)*>
<!--ELEMENT notSatisfied (action)*>
<!--ELEMENT global (commands)*>
<!--ELEMENT exit EMPTY>
<!--ATTLIST exit
id CDATA #IMPLIED
room CDATA #IMPLIED>

```

2.1.2 Parsing the XML to the world graph.

In order of the world to be used during runtime it needed to be converted into an Object. The chosen object type for the parsing was a graph of rooms for the room representation and a map of commands for the global commands. For parsing the XML, the use of the Jdom library has proven to be the best choice because of the tree form of the XML file it offers, and the access methods to the XML file. First the XML is read using a SAX builder, which parses the XML to a tree. Then each room is parsed on its own using the getChildren command which returns a list of the chosen element type.

3. GAME COMPONENT MODEL

The game model requires that there is an XML data of the world map along its inventory stored on a peer-to-peer model system. The player class is the main instantiation of the game execution.

3.1 Design Principles

Since the game revolves around commands and states of the world, its engine required a way to handle each command and send a response to it which must become visible to the player. For this functionality a number of classes are used to represent the game entities which are extracted from the XML data. These classes are a 1-1 mapping of the XML elements with specialized effects, and additional remote effects which are a way of indirect communication with the message service. The main class of the game engine, the player includes all other classes mentioned and handles interactions between them, and is responsible for communicating with other components of the game. It's connected to

other components using a request-response protocol, where any component which needs to communicate with the game engine, must send its input and wait for a response output, and handle the output. The player interface is the only means of changing the state of the player and the only way of verifying if a command is valid. It is a full representation of the game world as seen in a client, since the player class holds the player's details, the room in which he or she is, the entire game is mapped as a graph, the player state (inventory, active and finished quests, health points, and list of players with whom he or she is fighting), as well as a map to the global commands and their actions, as they are extracted from the XML data.

4. MESSAGING COMPONENT MODEL

One of the most important aspects of communication between clients is the broadcasting and reception of messages. Through messaging, the clients, update their local world states, and represent the states of other clients.

4.1 Design Principles

The publish-subscribe architecture pattern describes an abstract mechanism for loosely coupled, many-to-many communications between publishers of messages and subscribers who wish to receive messages [6]. The messaging component can accept any number of listeners and when a message is being received all of the listeners are being notified of this change and they can act upon the state change. The advantage of using this model is that we can add different components which can manipulate messages at any time just by having them subscribe to the message publisher. Also this separates the message component from the client component which enables the possibility of replacing the implementation of the message component with a different one without changing the subscribers. For the actual broadcast of messages there were several choices of technology including JGroups [7] and Jxta. All the messages are an extension of the AbstractMessage class, where a sender and a message string are defined, as well as a date and the message type. The AbstractMessage class is an abstract implementation of IMessage interface as defined below:

```
public interface IMessage extends Serializable
{
    public String getSender();
    public void setSender(String sender);
    public String getMessage();
    public void setMessage(String message);
    public void setDate(Date date);
    public Date getDate();
    public String getType();
}
```

Each type of message extends this class by adding any additional information it may require. Specifically, Location messages add the room id of the player's location as well as the player's faction. Chat messages add a scope and the receiving room id and/or player; cache update messages add a recipient who should update his or her cache. Chat

messages are extended by fight messages which are a special type of chat messages and use a specific message string for communication, and damage messages are a special type of fight messages which introduce a damage amount integer. To achieve this, several types of messages are used, each one sent when the appropriate effect is triggered, and are processed accordingly. There is also a distributed and a local cache used for sharing the user states that need to be shared. Sending Messages. Messages are sent through the JXTAConnection class using the send method. Messages are formed according to the commands the player enters, and are broadcast on the group. When a message has a limited scope, that message is discarded on reception if the receiving client is not a member of that scope. Receiving Messages. When a message is received, if its scope includes the receiving client, it is added to the appropriate history in the local cache and the message listeners are notified of an update of the local cache. Each receiving client then processes the message and responds to it either by representing it or sending a message of its own depending on the nature of the message. The types of messages are separated into state changing messages and player communication messages. State changing messages are Location messages, New User messages and Cache Update messages.

***Location messages:** The Location messages are sent when a client moves to a different room, so that the other clients update the state of the user locations they are using. They include the User name of the client, the Room in which the client is moving and the fraction of the user (whether angel or demon). When a location message is received by a client, that client updates the User Location class which it holds, and if the rooms affected affect the receiving client (either the room the player left from or the room the player enters is the room in which the receiving client is currently in) a room representation render follows.*

***New User messages:** A New User message is sent when a new user joins the game, sending an update of his name, location and fraction for the other users to represent in their local caches. The New User messages are processed differently depending on whether the client is a group leader or not. A new user has to get an update on the users who are already in the game, and so needs to get an updated distributed cache. When a client receives a new user message, it updates the user locations it keeps on its local cache, although if the receiving client is the group leader, it pushes an updated instance of the local user locations to the distributed cache, and sends a Cache update message to the new user to pull it.*

***Cache Update messages:** The cache update messages are messages that are sent when one or more players need to pull the distributed cache to get an updated view of the world. These messages are sent by the group leader to clients who don't have an updated view. When a client receives such a message, it must pull the distributed cache from the group handler and save it in its local cache. Player communication messages are separated into combat messages and chat messages. Fight messages include the Fight messages and the Damage messages.*

***Fight messages:** Fight messages are sent when a player initiates a fight with another player or when a player has received a fight challenge message. To start a fight, a player sends a fight challenge message to the target client. When a player receives such a message, it sends a Fight accepted message to the sending client which initializes the fight. When the client that sent the challenge receives an acceptance, the fight begins for it also. When a player's life total reaches zero during a fight, then a fight death message is sent to all the players with which the player was fighting, and a fight reward message is sent to the first player who initiated the fight. When a client receives a death message from another player, it stops fighting that player, and when a reward message is received, that player adds the reward to its inventory.*

***Damage message:** Damage messages are the backbone of combat. When a fight is initiated, all participating players send damage messages to the other players with whom they are fighting, over a two second interval. A damage message contains the receiving and sending user names, and the*

damage amount which is an integer less than or equal to 9. When a player receives a damage message, it subtracts that damage amount from its life total. The other form of communication messages are chat messages:

Chat messages: Chat messages are the means of social communication between players. They are separated into three scopes. The global scope is sent when a player uses the shout command. The room scope is sent when a player uses the say command and the personal scope is used when a player uses the whisper command. The global chat messages are represented in all clients, while the room messages are represented only by players who are in the same room as the sender, and personal messages are represented by the player set as a recipient, if that player is in the same room as the sender. All these types of messages are saved as a history in the local cache of each player. This cache contains a chat history, a new user history, a cache update history, and a location message history. The local cache communicates with the player by using the State changed event listeners and the state changed events. Whenever a change occurs in the local cache, the client is notified of the source of that change and can manipulate the messages which were received by reading them from the local cache.

5. GAME ENGINE

The game engine is the heart of the Maiar game software and is developed mainly on interfaces for easy extensibility of many actions and effects. It revolves around actions and effects that show in the 2D graphical user interface with skinning support. Based on a certain query of state of the room and an action inputted by the player/user will in turn then trigger an effect for the player like rendering a different room in the sequence of procedure described in the XML data. Let's look at the design principles of the engine component model.

Figure 3 Life cycle of remote command

5.1 Design Principles

The engine component (model) communicates with other components using a request - response architecture through its interface. That interface is the player class interface

which offers methods for retrieving trivial data from the player class, and two methods which can result in game state changes. These methods are the tryAction method, and the ExternalEffect method, both of which return a list of effects to be handled and represented to the GUI. The main difference between them is that the tryAction method takes as input a String command which is the player input command after it's been parsed by the command parser, while the ExternalEffect method takes as input a RemoteEffect, which is a type of effect which involves multiplayer interaction and is invoked by an incoming effect. In Figure 3, the life cycle of the remote interaction between GUI of peers is in Figure 3 and so is the internal life cycle of operational activities of local states described in Figure 4.

This architecture ensures that the engine component can work independently of either a multi player component or a GUI component, and can be used in a different type of game, which could have a different GUI and no multi player mode with just slight modifications. This also allows asynchronous communication with other players since the messaging engine works asynchronously from the GUI in Figure 5.

Figure 4 The life cycle of a local command.

Figure 5 Execution of Maiar P2P Game.

6. LOGGING COMPONENT MODEL

Maia Logging provides a facility for maintaining the development of the game in the development stage and for customers at production stage. Basically, the logging facility provides these core functionalities: Developer Diagnosis: The most obvious functional merit of logging is to help developers under development stages to be able to debug the application. Because a modular approach was used in designing the game, it is essential that each module is able to log its activities to help us to identify other problems that occur at runtime and other low-level libraries were used in the implementation, logs help locate problems in the application. End-Users and System Administrator Diagnosis: It is important that messages sent by a peer are transmitted successfully to the group, which is the core building block of the design. End-users can use logging activities in the application to convey the errors in the application to the System Administrators for system diagnosis. In practice, common end-user needed logs include system failure, memory issues and errors in the configuration files. With the Maia Logging, these logs are generated and can be configured to send email to the Maia development team.

6.1 Design Principles

There are a lot of logging frameworks out there, which each has different core functionality. The idea of designing Maia Logging over the other frameworks inclusively SIF4J [13], Log4j [9] and JDK logging requires a common interface to abstract the implementations of the frameworks from the implementation. The Logging compound component shows the three basic factory components each of which provides an access to a particular logging frameworks. The use of the abstract factory design pattern separates the details of the implementation of a set of objects from their general usage. The basic details of the factories are:

InternalLoggerFactory: *This creates an InternalLogger which changes the default factory implementation. This factory allows you to choose what logging framework Maia should use. The default factory is JdkLoggerFactory. You can change it to your preferred logging framework before other Maia classes are loaded. Each factory implementation only uses the new Instance method.*

JdkLoggerFactory: *This factory class extends the InternalLoggerFactory which uses the java.util.logging.Logger to provide the new instance of the ILogger class by passing the instance of java.util.logging.Logger to the JdkLogger class. The ILogger class provides access to basic logging functionality like debug messages, warn messages, info messages and error messages.*

Slf4jLoggerFactory: *This factory class extends the InternalLoggerFactory which uses the org.slf4j.Logger to provide the new instance of the ILogger class by passing the instance of org.slf4j.Logger to the Slf4jLogger class. ILogger interface provides the same functionality as that of the above but different implementation.*

Log4jLoggerFactory: *This factory class extends the InternalLoggerFactory which uses the org.log4j.Logger to provide the new instance of the ILogger class by passing the instance of org.log4j.Logger to the Log4jLogger class. ILogger interface provides the same functionality as that of the above but different implementation.*

7. SECURITY COMPONENT MODEL

At the architectural level, security boils down to understanding the precise security requirements for an application, and devising mechanisms to support them [6]. The most common security-related requirements [2] are:

Authentication: Applications can verify the identity of their users and other applications with which they communicate.

Authorization: Authenticated users and applications have defined access rights to the resources of the system. For example, some users may have read-only access to the application's data, while others have read-write.

Encryption: The messages sent to/from the application are encrypted.

Integrity: This ensures the contents of a message are not altered in transit.

Non-repudiation: The sender of a message has proof of delivery and the receiver is assured of the sender's identity. This means neither can subsequently refute their participation in the message exchange.

The architecture only supports authentication but the other three security-related requirements can be provided. New users to the system first create an account by registering their details and choosing user name and password. This information is stored in the database. Whenever users wish to play the game, they would supply user name and password and is checked against the one in the database, MySQL[11]. If the information provided matches with the information in the database, access is immediately granted. This ensures that no unauthorized user ever gets access to the system.

8. SOFTWARE QUALITY ATTRIBUTES

This section provides an overview of the software quality attributes that the application attains much to do with the choice of the technology. Quality attribute requirements are part of an application's nonfunctional requirements, which capture the many facets of how the functional requirements of an application are achieved [6]. The application captured the following major quality attributes (S-M-E-P) that the software Need:

Scalability(S): Scalability is one of the major concerns in client-server connection that is effectively dealt with in peer-to-peer connection. In client-server connection, increasing number of users on server may lead to total collapse of the server. On the other hand, peer-to-peer systems can have any number of users without the need to be concerned about central failure. System load is divided between clients. The Maiar game is designed using peer-to-peer model to take advantage of scalability that peer-to-peer connection provides.

Modifiability (M): Post-delivery maintenance [12] is necessary in product development and as such corrective, perfective and adaptive maintenance are things that require much attention and were taken care of in Maiar application.

Corrective Maintenance: Apache Maven as the software management and build tool helps us to modularly maintain dependencies to resolve coding faults.

Perfective Maintenance: But with the current architecture approach and using more industrial practice like design patterns, interface-implementation approach gives more simpler learning curve to make additional changes if so required.

Extensibility (E): XML is used in the system to create a map of the worlds. Commands and descriptions are also stored in XML file. Whenever a user issues a command, it is fetched from the XML file and executed. One of the main advantages of XML is extensibility.

Performance (P): P2P connection aid network performance, in that the bandwidth of the system increases as each peer arrives the bandwidth capacity of the system increases as more peers arrive: each new peer requests service from, but also provides service to, the other peers. The network can thus spontaneously adapt to the demand by taking advantage of the resources provided by every peer. We might have to performed more test to assess the performance of this in relation to the application.

9. DEPLOYMENT

The executable program will be installed on the peer system after has successfully registered with the Maiar Server Web Application. For the client peer to work, it requires some dependencies which are just Java Executable jar files as shown in Figure 6. A Peer requires the Java Runtime Environment (JRE) installed on the system to be able to play the game, Maiar. There is an optional installation process for Peers that does not have the JRE installed on their systems and this is bundled with the Game installer deployed and hosted on the Application Server. Peer Dependencies: For a peer to successfully run the Maiar Game, these are the required artifacts on their system:

- **Logging.jar:** This artifact is the logging component of the application. It is required to provide the basic logging functionality at the peer side.
- **Model.jar:** This artifact contains the domain model of the game
- **Client.jar:** This artifact contains the Graphical User Interface, which is the Swing Application with interactive response and user experience.
- **Messaging.jar:** This artifact is the most essential component of the peer dependency because Maiar server classes are placed.

Figure 6 Deployment View

CONCLUSION. This work has thoroughly detailed the design and architecture of Maiar game software mainly in terms of the game engine design principles, game modeling of actions/effects, messaging design of the message exchange component model, security model of the game application, logging component for application messages (warning, debug, information, etc) software attributes needed in the requirement process and deployment view of the game software in general.

References

- [1] K. A. Berman and J. L. Paul. *Algorithms: Sequential, Parallel and Distributed*. Thomson Course Technology, Boston, Massachusetts, 2005.
- [2] D. A. Blyth, D. D. Cunliffe, and D. I. Sutherland. *Security analysis of xml usage and xml parsing*. *Computers and Security*, 2003.
- [3] B. Daum. *Modeling Business objects with XML Schema*. Morgan Kaufmann Publishers, Germany, 2003.
- [4] T. Faison. *Event-based Programming: Taking Events to the Limit*. Apress, May 2006.
- [5] R. Garg, V. K. Garg, and Y. Sabharwal. *Efficient algorithms for global snapshots in large distributed systems*. *IEEE Transactions on Parallel and Distributed Systems*, 99(RapidPosts):620–630, 2009.
- [6] I. Gorton. *Essential Software Architecture*. Springer-Verlag Berlin Heidelberg, Germany, 2006.
- [7] JGroups. <http://www.jgroups.org/>, 2010.
- [8] JXTA. <https://jxta.dev.java.net/>, 2010.
- [9] Log4j. <http://logging.apache.org/log4j/1.2/index.html>, 2010.
- [10] Maven. <http://maven.apache.org/>, 2010.
- [11] MySQL. <http://mysql.com/>, 2010.
- [12] S. R. Schach. *Object-Oriented and Classical Software Engineering*. McGraw Hill, Germany, 2005.
- [13] SLF4J. <http://www.slf4j.org/>, 2010.
- [14] R. Steinmetz and K. Wehrle. *P2p systems and applications*. LNCS 3485, pp. , 2005. Springer-Verlag Berlin Heidelberg, 15(5):9–16, November 2005.
- [15] C. M. S.-M. Tim Bray, Jean Paoli and E. Maler(Eds). *Extensible markup language (xml) 1.0 (fifth edition)*. W3C Recommendation, 26, November 2008.
- [16] Tomcat. <http://tomcat.apache.org/>, 2010.

Frank Appiah is a professor of computer engineering, software engineering and informatics at University of London, KCL. Frank Appiah is holding both academic and teacher positions after the programmes of study; Bsc. (Bachelor in Computer Engineering, Kwame Nkrumah University) in 2009, Msc. (Advanced Software Engineering, King's College London) Level in 2009/10 and PGCerts/PGDip (educational skills, postgraduate student, King's/IEEE) diplomatic level in 2010. Prof. Appiah developed p2p game software called Maiar Game in Java programming language. Appiah is a member of the British Academy of Engineering since 2010/2011 and holds the 'ING' title from Ghana Institute of (Software) Engineering. He is a member of IEEE Computer. Contact him at appiahnsiahfrank@gmail.com