# Introduction (Alex Craig)

This report is intended as a final report of progress by the RoboWars SYSC 4907 engineering project team. The RoboWars project aims to further develop solutions in the area of remote robotics control systems. Control systems for remote robotics available today often require dedicated and specialized hardware to interface with the robotics, and may provide only rudimentary control functionality. By exploiting widely available consumer hardware and open source software, the RoboWars project aims to create a feature rich robotics control system that can be easily adapted to any Bluetooth capable robotics system.

Developing a robotics control system on a widely available mobile platform provides developers with greater accessibility and reduced development time. Using smartphones to remotely control devices, robotic or not, eliminates the need for extra time and development resources to be spent on designing and building a specialised hardware platform to run the control software on. Also, with smartphones already in widespread use by the public, it is simple and intuitive for a user to download, install, and immediately use the application to remotely control their device. This provides developers with a well established and easily accessible customer base.

In addition to avoided hardware development costs, smartphones provide a powerful platform which can support a wide range of features to enhance the usability of the controlled robotics. Smartphones are typically equipped with high resolution touch screens, Wi-Fi and Bluetooth communication capabilities, accelerometers, and dedicated graphics processors. These capabilities allow a robotics system to support enhanced features that may not be available with simpler dedicated hardware, such as live video streaming, three dimensional graphical rendering, tilt or touch screen based movement control, and high bandwidth data exchange to support advanced robot behaviour.

The RoboWars project has two primary objectives:

* The first project objective is to develop a robotics control system which is both intuitive to use and is implemented on a mobile platform that is widely available and used by the public. Specifically, the project targets mobile smartphones running the Android operating system as the client platform.
* The second project objective is to experiment with the combination of live video and virtually generated, overlaid imagery to enhance the ease of use and feature set of a robotics control system. This technology is commonly referred to as augmented reality.

To combine these objectives, the project aims to create a solution which allows two remotely controlled LEGO Mindstorm NXT 2.0 robots to share and interact with a simple virtual world which will be rendered in three dimensions and overlaid on a live video feed to be displayed to the remote operators. Users can connect to the system from smartphones anywhere where an internet connection is available, and use the robots to play simple virtual games supported by augmented reality overlays.

The following components of the proposed solution have been successfully implemented:

* Centralized server software which manages connections, propagates commands and messages, and records client state from Android clients (via standard IP) and Mindstorm NXT 2.0 robotics kits (via Bluetooth).
* Mindstorm NXT 2.0 Intelligent Brick software which supports duplex Bluetooth communication, remote execution of commands received from the central server, dead reckoning position tracking and error correction, and real time position and heading data transmission to the central server.
* Server side virtual world implementation which is capable of asynchronously updating robot positions, filtering client commands based on validity against the current game state, and generating outgoing robot commands to ensure robot movement does not violate the virtual game state.
* Real time video streaming from a USB webcam connected to the central server to connected Android clients using a custom UDP protocol.
* Android client software which supports OpenGL rendering of incoming real time video streams, as well as rudimentary three dimensional graphics rendering (simple textured polygons).

The remainder of this report will detail the development methodology of the RoboWars project, and the technical specifications of the implemented solution. First, the requirements for the project will be discussed, as well as the engineering principles and development practices used to implement the proposed solution. Next, the technical details of the implementation will be discussed, followed by a discussion of the testing methodology employed and results obtained. Finally conclusions and recommendations for alternative implementations and further work will be presented.

# The Engineering Project

# Background and Terminology

# Requirements (Alex Craig)

The RoboWars project was an original idea proposed by the project team, and as such no external client existed to participate in requirements elicitation. To ensure that the project scope and requirements were clearly defined requirement elicitation was performed within the project group, and the requirements were documented through the collection of use cases shown in Table 4.1. A complete use case specification was produced, which has been included as Appendix A. This document provides a detailed description of the actors which interact with the system, as well as a formal specification for each use case. In addition, use case realizations (in the form of sequence diagrams) were also produced for selected use cases to aid in early design work. Theses sequence diagrams are included as Appendix B.

Table .1: RoboWars Use Case Names and Descriptions

|  |  |
| --- | --- |
| **Name** | **Description** |
| **Configure wired camera** | The server administrator selects the camera to use from a list of detected cameras, and configures the position, facing, and field of view of the selected camera. |
| **Display local video feed** | A live video feed of the “arena’ containing the remote robots is read from the attached camera and displayed through the server administration GUI. |
| **Reset robot positions** | The server administrator forces a reset of the robot positions in the system’s internal state. This should be performed if the system’s positioning information for a robot becomes out of sync with the robot’s actual position for any reason. |
| **Display simplified game state** | The server administrator triggers the display of a simple 2D rendering of the current game state which is visible in the server administration GUI. |
| **Register robot for remote control** | A remote robot connects and registers with the system. |
| **Unregister robot for remote control** | A remote robot unregisters from remote control and disconnects from the system. |
| **Update robot position** | A robot asynchronously updates the system to reflect its current position. |
| **Join waiting lobby** | A mobile client initializes a connection to the server. |
| **Send chat message** | A mobile user sends a chat message to be displayed to all other connected mobile users. |
| **Opt-out of robot control** | A mobile user who wishes to view rather than control the remote robots may choose to opt-out of robot control and become a pure spectator. |
| **Select game type** | A player selects the type of game to play prior to starting a new game. |
| **Pair play to robot for remote control** | A specific player is paired to a specific robot for the duration of a single game. Any commands from the player will control only the robot paired to the specific player. |
| **Launch game** | A player in the lobby initiates a game. The lobby is replaced with an augmented video feed, and players are paired to robots for remote control. Virtual gameplay begins. |
| **Display augmented video** | The live video feed of the camera overlooking the robot arena is combined with the internal game state representation to produce virtual elements overlaid on the live video feed. |
| **Move robot** | A mobile player issues a command for a robot to move, which is passed to the robot by the system. |
| **Fire virtual projectile** | A mobile player issues a command to fire a virtual projectile, which is created and simulated in the system. |
| **End game** | A game in progress is ended and all connected mobile users are returned to the game lobby. Any paired robots are freed and considered unpaired. |

# Overall System Architecture (Alex Craig)

The RoboWars system is designed as a centralized system, in which both robot and Android clients connect to a central server which manages the system state and propagates information between clients as required. The RoboWars system is distributed over three types of nodes:

**Central Server:** This node represents standard consumer desktop hardware, with the minor addition of a Bluetooth dongle to support communication with the Mindstorm NXT 2.0 robotics kits. A Model-View-Controller architecture is used to separate the main server application into three subsystems which are all deployed on this node. The controller subsystem is responsible for managing robot and Android client connections, broadcasting the real time video stream, managing the server lobby (which entails broadcasting chat messages, and launching / terminating model instances), propagating robot position changes to the model, and continually triggering model physics updates to provide real time physics in the virtual world. The model component is dynamically constructed whenever a new game is initialized, and is responsible for storing the current game state (including virtual simulated entities), broadcasting state changes to the other subsystems, and filtering all client commands to ensure that virtual game state will not be violated. The view component observes both the model and controller components, and provides an admin interface through which the server and camera settings are configured, the chat lobby is monitored, and the current game and client connection state is displayed.

**Robot:** The robot node represents a Mindstorm NXT 2.0 Intelligence Brick. Up to two connected robot nodes have been tested, although the implementation is designed to support an arbitrary number of robots. The robot client is implemented as a single subsystem which is responsible for local position tracking and remote execution of commands from the central server. Although the RoboWars implementation is designed with a custom Mindstorm NXT 2.0 client in mind, any Bluetooth capable robotics kit could be adapted to this purpose.

**Android:** The Android node represents any Android 2.2 smartphone. Up to two connected Android nodes have been tested, although the implementation is designed to support an arbitrary number of Android clients (including spectators who do not directly control a robot). The Android client is implemented as a two subsystems: the Android client and the game model (which is shared with the server implementation). The Android client subsystem is responsible for transmitting chat and control messages to the server, receiving and rendering the real time video stream, rendering OpenGL graphics, passing orientation and touch screen input to the central server, and receiving game state updates to be integrated into the model. The model subsystem is continually updated to match the current server state, and is used to determine the positions of entities for use with OpenGL rendering.

A deployment diagram demonstrating the distribution of subsystems among nodes is shown in Figure 5.1.

Figure ‑: A deployment diagram of the complete RoboWars system. Note that multiple instances of the Robot and Android nodes can be serviced by a single Central Server node.



# Server Implementation

The central server is the most complex node in the RoboWars system, manages all communication flows between the Android clients and robots. See Appendix D for class diagrams of the server implementation.

## Controller - User and Robot State Management and Data Propagation (Alex Craig)

The central component of the controller with respect to user and robot state management is an instance of the ServerLobby class. ServerLobby is a passive class that is not concerned with any real time aspects of gameplay, but rather manages lists of references to instances of UserProxy and RobotProxy (each of which represents a connected client), and controls the launching and termination of real time gameplay. The ServerLobby uses an event model to allow any class which implements the ServerLobbyListener interface to receive events whenever a player or robot joins or leaves the server, whenever a chat message is received, or whenever a game is launched or terminated. In the current implementation, this functionality is used to allow the view subsystem to display lists of connected clients and the server chat lobby.

To manage incoming Android client connections, the controller uses an instance of TcpServer, an active class which runs a separate thread to continually accept incoming TCP connections on a specified IP address and port. Whenever a new connection is received, the reference to the client’s socket is used to construct a UserProxy instance. Each UserProxy is also an active object which runs a dedicated thread to continually read new messages from the client’s socket. Once a UserProxy is generated, a handshake procedure is carried out to ensure the version of the RoboWars client is compatible with the server. The client must also provide a unique username, and the client will be required to select a new username if the username is already in use. Once the handshake procedure is complete, the user is registered with the server’s main lobby, and users can interact with the server lobby by sending chat messages, changing the game mode, changing their own ready or spectator status, and initiating game launches. In addition, once the handshake is complete all further communication is achieved by sending serialized event classes rather than raw strings and the UserProxy registers as a listener on the ServerLobby. In this manner, instances of UserProxy listen on the main ServerLobby instance, and propagate events to their respective clients. This could have been implemented without an event model (as the ServerLobby stores references to all connected instances of UserProxy and could call functions directly), but it was decided that an event model was preferable as this method avoids coupling with the view subsystem and sets out a standard communication path for both the view subsystem and instances of UserProxy.

Robot connections are handled similarly to Android clients, but rather than awaiting incoming connections, the server actively attempts to discover compatible Mindstorm NXT 2.0 clients within Bluetooth range. This is accomplished using the PC libraries of LeJOS (“Java for LEGO Minstorms”) [ac1]. The functionalities of LeJOS in general will be discussed further in Section 7 (“Robot Client Implementation”). LeJOS provides a simple interface to enable either Bluetooth or USB communication with Mindstorm NXT 2.0s running LeJOS firmware. First, an NXTComm object must be generated by supplying a static method of the NXTCommFactory class with a constant specifying whether USB or Bluetooth communication is desired. In the case of Bluetooth communication, the NXTCommFactory will automatically select a compatible underlying implementation from one of several supported libraries based on the underlying operating system. RoboWars has been implemented and tested exclusively on Windows, and in this case LeJOS will select Bluecove [ac2] as its underlying Bluetooth implementation. Bluecove is an open source library implementation of the standard Java Bluetooth API specified in JSR-82. However, the details of the underlying implementation are largely unimportant for RoboWars, as LeJOS provides methods to write and read from Bluetooth using the standard Java input/output stream API once a connection has been established. It is worth noting that Bluetooth discovery process blocks for a significant duration (approximately 5 seconds for each detected Bluetooth device), and the BluetoothServer class is implemented such that robot redetection always runs in a dedicated thread. Once a connection is established, a RobotProxy instance is created and registered with the ServerLobby. RobotProxy is similar to UserProxy in that it is an active class that continually reads the Bluetooth input stream to receive position updates from its associated robot. When a game is in progress this position information will be propagated into the game model, and robot commands will be propagated through the RobotProxy to its associated robot.

Once both instances of UserProxy and RobotProxy are registered with ServerLobby, a game can be launched. Game launch requests must be issued by an Android client, and a UserProxy propagates the request to the ServerLobby. The ServerLobby will ensure that enough (non-spectator) players and robots are registered for the selected game type, and will generate a new GameController instance to represent the new real time game. The GameController is an active class which generates an encapsulated instance of the game model, and runs a thread to continuously update the model’s physics implementation. Users are selected for pairing with robots by the ServerLobby (based on which users have been paired most recently), and the ServerLobby generates an instance of ControlPair for each pairing. ControlPair class is a passive class used primarily for data encapsulation which stores a reference to a single UserProxy and RobotProxy. Instances of ControlPair are registered with the GameController, and once all control pairs have been registered with the GameController additional players are added as spectators. When this is complete, the game is launched. This generates a ServerLobbyEvent which signals to clients that a game has begun and OpenGL rendering should be displayed.

Once game play has been initialized, instances of UserProxy send tilt vectors and button input to the GameController where the data is used to generate instances of RobotCommand based on the control scheme specified by game mode in use. These commands are validated against the game model to ensure that game state will not be violated, and are then serialized and written to the user’s paired robot through the RobotProxy reference contained in the instance of ControlPair. Whenever model’s game state changes, the GameModel generates GameEvent objects which are captured by the listening GameController, and passed to all connected instances of UserProxy.

## Controller – Real Time Video Streaming (Alex Craig)

### Alternative Solution – RTP/RTSP Streaming

Ideally, the real time video streaming in the RoboWars should use an established and well defined protocol. The first attempts to implement video streaming in RoboWars made use of the Real-Time Transport Protocol (RTP) for data transmission, and Real Time Streaming Protocol (RTSP) for control signals. This functionality is well supported by the standard Android library, which provides a MediaPlayer class specifically designed to receive and display and RTP/RTSP stream [ac3]. To test this functionality, the Android client for RoboWars was modified to make use of the MediaPlayer class to accept and display an RTP/RTSP video stream.

The server side generation of the RTP data stream was partially implemented using the “Freedom for Media in Java” (FMJ) project [ac4], which provides an open source implementation of the standard Java Media Framework API [ac5] for capturing, playing, and streaming media. JMF provided the means to easily generate an RTP data stream, but further investigation determined that alternate means would need to be used to implement an RTSP server to generate the required control signals expected by the Android MediaPlayer. Rather than continue with server implementation, the open source VideoLAN (VLC) project was used to generate video streams purely to test the Android client implementation.

Through this testing, it was determined that the capabilities of the Android MediaPlayer implementation were insufficient for the needs of the RoboWars project. Although the video stream could be reliably displayed, an unacceptable delay of approximately 10 seconds occurred between the transmission and rendering of the video stream. Further testing determined that the cause of this delay was the buffering behaviour of the MediaPlayer implementation. The MediaPlayer class is intended to be used for the streaming of relatively static media (ex. YouTube videos) rather than real time streams with low delay requirements. The implementation of MediaPlayer buffers video based on a fixed time of video, rather than data size, and does not provide any external API to modify this behaviour. Regardless of the resolution or bit rate of the video stream generated by VLC the ten second delay on the client side remained constant. Further investigation of the Android source code revealed that this buffering functionality is implemented at the level of phone firmware. Modifications to the Android firmware would require that all users of RoboWars install custom firmware on their phones, and would limit RoboWars clients to phones that have had firmware specifically created for RoboWars. This would have significantly reduced the generality and potential market for RoboWars, and as such further modifications were not attempted at the firmware level.

### Implemented Solution – Custom UDP Protocol

In order to provide real time video streaming with an acceptably low delay a custom protocol for data transmission was required. The use of the FMJ project to interface with the webcam was discarded (as the RTP stream generation was no longer required), and replaced with the simpler LTI-Civil library [ac6] (which is also used internally within FMJ). LTI-Civil is an open source library which provides a simple API for capturing images from video devices entirely independently of the JMF API.

Video streaming is implemented through two classes in the controller subsystem: CameraController and MediaStreamer. The CameraController class is responsible for storing the position, heading, and field of view of a specific camera, as well as opening and making available the instance of CaptureStream interface which is used to read images from the camera. The MediaStreamer class is responsible for discovering available cameras, maintaining a valid list of instances of CameraController, and maintaining a list of users that video packets should be served to. The MediaStreamer also provides methods to select an active camera from the list of CameraControllers, and methods to start and stop the active capture stream. The MediaServer is registered as a listener on the main ServerLobby instance, and automatically adds and removes users to its list of clients to serve whenever a user joins or leaves the main lobby. In addition, the MediaStreamer also receives game launch and termination events from the main ServerLobby, and the video stream is launched or terminating along with real time gameplay.

To serve video, the MediaStreamer must receive a valid instance of CaptureStream from the active CameraController. At this point, the observer pattern is used to read frames from the webcam. MediaStreamer implements the CaptureObserver interface, which allows it to register as a listener on the active CaptureStream instance. Once the stream has been launched, the onNewImage() function of MediaStreamer will be called by LTI-Civil at a fixed rate (determined before the CaptureStream is launched).

Video is transmitted to clients as a series of individual frames encoded as JPEGs. The onNewImage() function of MediaStreamer receives frames from the webcam in an unspecified format internal to LTI-Civil which is converted into an instance of BufferedImage from the Java standard library using a static method of the AWTImageConverter class (provided by LTI-Civil). Once the frame is in BufferedImage format, a static method of the ImageIO class (from the Java standard library) is used to encode the frame as a JPEG and write the result to a byte array. This byte array is split into segments of a configurable size (set via a static constant in MediaStreamer), and each segment is packaged into a separate UDP DatagramPacket. Each packet carries three fields in addition to the video data payload:

* **Frame number**: Specifies which frame this packet belongs to. In the current implementation this field alternates between 1 and 0, as it is unlikely that an individual segment will arrive more than a full frame out of sequence (and with the next segment number expected by the client). Even if this does occur it will only result in a dropped frame, which is unlikely to be noticed by the user.
* **Segment number**: Specifies which segment of the current frame this packet carries. This field starts at 0 for the first segment of each new frame, and is incremented by 1 for each packet sent.
* **Last segment**: A Boolean field which is set to true only on the last packet of a given frame.

These generated packets are transmitted to all connected clients through UDP unicast on a specified port (the main server TCP port incremented by 1). UDP multicast would be preferable for this application as it reduces the overall bandwidth requirements (as packets are duplicated as required at the network level rather than at the server), and this method was originally implemented. However, the HTC Desire smartphones utilized by the project for testing were determined to be incapable of receiving multicast or broadcast packets, and as such the usage of UDP multicast could not be properly verified. This shortcoming is not officially documented, but is well corroborated by other developer reports [ac7]. Although UDP unicast is less bandwidth efficient, its usage allowed for the project to test video streaming functionality and perform demos with the available hardware.

The Android video client only supports the decoding of incoming video frames, and does not exchange any further information with the server (over UDP). A byte buffer is kept, and the video data from each incoming packet is appended to the buffer whenever a packet is received that matches the next expected frame and sequence number. When a packet with the “last segment” field is received, the buffer is decoded into the standard Android image format using the BitmapFactory class provided by the Android standard library. This bitmap is then passed to the OpenGL renderer to be loaded as a texture. The client’s buffer is cleared and the expected incoming frame and sequence numbers are reset whenever a new packet is received with a segment number of 0 (indicates the first segment of a new frame).

## View – Administration Panel and Camera Configuration (Alex Craig)

The administration panel uses standard Swing libraries to provide a GUI for the administrator to configure the server. The main frame of the administration panel is implemented in the AdminView class, which subclasses JFrame. The AdminView class implements ServerLobbyListener and listens for events from the main ServerLobby instance to update the connected user/robot lists and the main chat panel. The AdminView also contains a reference to the BluetoothServer instance, which is used to manually trigger robot redetection. This was originally implemented as a periodic function that did not require administrator interaction, but the period redetections were found to interfere with communication with robots during gameplay. The AdminView also generates an instance of CameraSelectionView which contains a reference to the MediaStreamer instance, and allows the administrator to configure the camera settings for the server.

## View – 2D Display (Alex Dinardo)

## Model - Virtual World Model Implementation (Alex Dinardo)

## Logging (Alex Craig)

RoboWars makes extensive use of the Apache Log4j open source logging library [ac8] in order to aid with debugging and performance testing. This carries a number of advantages over console logging, including:

* All logging statements are tagged with the thread identifier of the thread which initiating the logging, the class name of the class which initiated the logging, and the current system time (in milliseconds). This data is very useful to have in a standardized format when debugging and acquiring performance metrics.
* Up to 6 independent levels of logging are supported (TRACE, DEBUG, INFO, WARN, ERROR and FATAL), each of which can be configured separately.
* Logging configurations are modified through a configuration file which is read at runtime. Logs can be directed to console or file, and these settings can be set independently for each logging level (by default, RoboWars logs DEBUG messages to file, and INFO and ERROR messages to both console and file). Suppressing log output can be done through runtime configuration, rather than modification of source code.

# Robot Client Implementation (Mike Wright)

# Android Client Implementation

# Testing

# Conclusion (Mike Wright)

# References

[ac1] “LeJOS, Java for LEGO Mindstorms”, http://lejos.sourceforge.net/ Accessed Mar. 15, 2011.

[ac2] “Bluecove JSR-82 Project”, http://bluecove.org/ Accessed Mar. 15, 2011.

[ac3] “MediaPlayer | Android Developers,” http://developer.android.com/reference/android/media/MediaPlayer.html Accessed Mar. 15, 2011.

[ac4] “FMJ”, http://fmj-sf.net/ Accessed Mar. 15, 2011.

[ac5] “Java SE Desktop Technologies – Java Media Framework API (JMF),” http://www.oracle.com/technetwork/java/javase/tech/index-jsp-140239.html Accessed Mar. 15, 2011.

[ac6] “LTI-Civil,” http://lti-civil.org/ Accessed Mar. 15, 2011.

[ac7] “Multicast on Android 2.2,” http://stackoverflow.com/questions/3623143/multicast-on-android-2-2 Accessed Mar. 15, 2011.

[ac8] “Apache log4j 1.2,” <http://logging.apache.org/log4j/1.2/> Accessed Mar. 15, 2011

# Appendix A – Schedule Gantt Chart

# Appendix B – Use Cases

# Appendix C – Use Case Realizations (Sequence Diagrams)

# Appendix D – Class Diagrams