A TEAM COMPUTING IMPLEMENTATION ON THE ANDROID PLATFORM

by

Xiang Shen
A Project
Submitted to the
Graduate Faculty
of
George Mason University
in Partial Fulfillment of
The Requirements for the Degree
of
Engineer Degree
Information Technology

Committee:

Dr. Joao Sousa, Project Director
Dr. Sam Malek, Committee Member
Dr. Alexander Brodsky, Committee Member
Dr. Stephen Nash, Senior Associate Dean
Dr. Kenneth S. Ball, Dean, Volgenau School of Engineering

Date: 5/03/2013
Spring Semester 2013
George Mason University
Fairfax, VA
A Team Computing Implementation on the Android Platform

A thesis submitted in partial fulfillment of the requirements for the degree of Engineer
Degree Information Technology at George Mason University

by

Xiang Shen
Master of Science
Maharishi University of Management 2003
Bachelor of Engineering
University of Science & Technology, Beijing 1997

Director: Joao Sousa
Department of Computer Science

Spring Semester 2013
George Mason University
Fairfax, VA
DEDICATION

This work is dedicated to my loving wife, Yi, and my two wonderful children Anne and Ray.
ACKNOWLEDGEMENTS

I would like to thank my many friends, relatives, and supporters who have made this project happen. My wife, Yi, assisted me in my research. Drs. Sousa, Malek, and Brodsky of my committee were of invaluable help.
# TABLE OF CONTENTS

<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>List of Tables</td>
<td>vi</td>
</tr>
<tr>
<td>List of Figures</td>
<td>vii</td>
</tr>
<tr>
<td>Abstract</td>
<td>ix</td>
</tr>
<tr>
<td>Introduction</td>
<td>1</td>
</tr>
<tr>
<td>Related Work</td>
<td>6</td>
</tr>
<tr>
<td>Use Cases</td>
<td>9</td>
</tr>
<tr>
<td>Team Management</td>
<td>10</td>
</tr>
<tr>
<td>Players Interaction in a Team</td>
<td>15</td>
</tr>
<tr>
<td>Self-healing</td>
<td>20</td>
</tr>
<tr>
<td>System Architecture</td>
<td>25</td>
</tr>
<tr>
<td>Data Model</td>
<td>27</td>
</tr>
<tr>
<td>Design &amp; Implementation</td>
<td>28</td>
</tr>
<tr>
<td>Overview</td>
<td>28</td>
</tr>
<tr>
<td>Editor</td>
<td>31</td>
</tr>
<tr>
<td>Player</td>
<td>33</td>
</tr>
<tr>
<td>Trigger Evaluation</td>
<td>35</td>
</tr>
<tr>
<td>Use Case Implementation</td>
<td>38</td>
</tr>
<tr>
<td>$B$ self-Healing</td>
<td>47</td>
</tr>
<tr>
<td>Alternative Implementation</td>
<td>57</td>
</tr>
<tr>
<td>Introduction of JGroups</td>
<td>57</td>
</tr>
<tr>
<td>Using JGroups</td>
<td>58</td>
</tr>
<tr>
<td>Summary of JGroups</td>
<td>63</td>
</tr>
<tr>
<td>System Evaluation</td>
<td>64</td>
</tr>
<tr>
<td>Evaluation Criteria</td>
<td>64</td>
</tr>
<tr>
<td>Evaluation Configuration</td>
<td>64</td>
</tr>
<tr>
<td>Communication Efficiency</td>
<td>65</td>
</tr>
</tbody>
</table>
# LIST OF TABLES

<table>
<thead>
<tr>
<th>Table</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>Table 1 Use Case Actors</td>
<td>9</td>
</tr>
<tr>
<td>Table 2 Editor Classes</td>
<td>32</td>
</tr>
<tr>
<td>Table 3 Player Classes</td>
<td>34</td>
</tr>
<tr>
<td>Table 4 Trigger Engine Classes</td>
<td>37</td>
</tr>
<tr>
<td>Table 5 QR Code Example</td>
<td>39</td>
</tr>
</tbody>
</table>
# LIST OF FIGURES

<table>
<thead>
<tr>
<th>Figure</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>Figure 1 An Example of TeC Team</td>
<td>3</td>
</tr>
<tr>
<td>Figure 2 Use Cases for Team Management</td>
<td>10</td>
</tr>
<tr>
<td>Figure 3 Use Cases for Players Working in a Team</td>
<td>15</td>
</tr>
<tr>
<td>Figure 4 Use Cases for Self-healing</td>
<td>20</td>
</tr>
<tr>
<td>Figure 5 System Architecture of a TeC Environment</td>
<td>25</td>
</tr>
<tr>
<td>Figure 6 Data Model for the Relational Database</td>
<td>27</td>
</tr>
<tr>
<td>Figure 7 A &quot;Simple XML&quot; Example</td>
<td>30</td>
</tr>
<tr>
<td>Figure 8 Editor Class Diagram</td>
<td>33</td>
</tr>
<tr>
<td>Figure 9 Player Class Diagram</td>
<td>35</td>
</tr>
<tr>
<td>Figure 10 Trigger Expression in Activity Sheet</td>
<td>36</td>
</tr>
<tr>
<td>Figure 11 Form of triggering conditions[3]</td>
<td>36</td>
</tr>
<tr>
<td>Figure 12 Trigger Engine Class Diagram</td>
<td>38</td>
</tr>
<tr>
<td>Figure 13 Invoke ZXing to Scan QR Code</td>
<td>39</td>
</tr>
<tr>
<td>Figure 14 Obtain Scan Result from ZXing</td>
<td>40</td>
</tr>
<tr>
<td>Figure 15 Add Device to TeC Environment Sequence Diagram</td>
<td>41</td>
</tr>
<tr>
<td>Figure 16 Discover Players Sequence Diagram</td>
<td>41</td>
</tr>
<tr>
<td>Figure 17 Deploy Team Sequence Diagram</td>
<td>42</td>
</tr>
<tr>
<td>Figure 18 Retire Team Sequence Diagram</td>
<td>43</td>
</tr>
<tr>
<td>Figure 19 Receive Briefing Sequence Diagram</td>
<td>44</td>
</tr>
<tr>
<td>Figure 20 Receive Retiring Sequence Diagram</td>
<td>44</td>
</tr>
<tr>
<td>Figure 21 Receive Input Sequence Diagram</td>
<td>45</td>
</tr>
<tr>
<td>Figure 22 Evaluate Trigger Sequence Diagram</td>
<td>46</td>
</tr>
<tr>
<td>Figure 23 Send Output Sequence Diagram</td>
<td>47</td>
</tr>
<tr>
<td>Figure 24 Self-healing 1: Request for a Healer</td>
<td>48</td>
</tr>
<tr>
<td>Figure 25 Self-healing 2: Reply to Healer Request</td>
<td>49</td>
</tr>
<tr>
<td>Figure 26 Self-healing 3: Briefing Healer</td>
<td>50</td>
</tr>
<tr>
<td>Figure 27 Self-healing 4: Bonding Complete</td>
<td>51</td>
</tr>
<tr>
<td>Figure 28 Self-healing 5: Losing a Player</td>
<td>52</td>
</tr>
<tr>
<td>Figure 29 Self-healing 6: Search New Player &amp; Healer</td>
<td>52</td>
</tr>
<tr>
<td>Figure 30 Self-healing 7: New Player Appears</td>
<td>54</td>
</tr>
<tr>
<td>Figure 31 Self-healing 8: New Player Brief Healer</td>
<td>54</td>
</tr>
<tr>
<td>Figure 32 Self-healing 9: Update Stale Variables</td>
<td>55</td>
</tr>
<tr>
<td>Figure 33 Self-healing 10: Bonding Complete Again</td>
<td>56</td>
</tr>
<tr>
<td>Figure 34 XML Property File for JGroups</td>
<td>60</td>
</tr>
<tr>
<td>Figure 35 Setup JGroups Programmatically</td>
<td>61</td>
</tr>
</tbody>
</table>
Figure 36 Create Channel and Send/Receive Messages with JGroups.......................... 62
Figure 37 JGroups Callback Used for Self-healing.................................................. 63
Figure 38 Another TeC Example............................................................................... 65
Figure 39 Power Consumption for a TeC Player...................................................... 67
Figure 40 Power Consumption for a TeC Player Using JGroups............................... 68
Figure 41 CPU & Memory for a TeC Editor............................................................. 69
Figure 42 CPU & Memory for a TeC Editor Using JGroups...................................... 69
Figure 43 CPU & Memory for a TeC Player............................................................. 70
Figure 44 CPU & Memory for a TeC Player Using JGroups...................................... 71
ABSTRACT

A TEAM COMPUTING IMPLEMENTATION ON THE ANDROID PLATFORM

Xiang Shen

George Mason University, 2013

Project Director: Dr. Joao Sousa

This thesis describes a software system that was developed based on the Team Computing (TeC) model. TeC belongs to Ubiquitous Computing, which emphasizes the coordination of teams comprised of software components, devices, and human operation. This particular system is implemented on the popular Android platform using Java. It allows the teams designed by end users to be deployed and operate on Android devices. The system also implements a decentralized, lightweight protocol for self-healing.
INTRODUCTION

Nowadays, the computing power of many embedded systems and mobile devices has increased to such a level that it can easily surpass a powerful PC manufactured ten years ago. Operating systems for mobile devices such as iOS and Android have also become quite popular and provide a rich feature set. These trends make Ubiquitous Computing[1] more appealing and viable than before.

Team Computing (TeC) is a framework for end-user design, deployment, and evolution of applications for Ubiquitous Computing. Systems in TeC act as teams of elements with no central control: “elements play their roles autonomously, and system behavior is emergent”[2].

As described in prior work (Sousa et al.[3]), TeC is based on the concept of a team as a collection of players that collaborate to achieve a purpose. Running software components, network-enabled devices, and people may become players in a team.

The computation carried out by a team results from the joint effect of the activities performed by the individual players. These activities are coordinated by asynchronous communication of data and events. Activities generalize the computational constructs of services and processes, as well as the functions performed by devices, and the actions performed by humans. Activities may be of short duration with a discrete output, like services; they may be long lasting with a succession of inputs and outputs, like processes;
or they may last for years and never produce an output, like the function of a smoke detector.

**Activity types** capture generic knowledge about activities, such as possible inputs and observable properties. Inputs may consist of events and data streams. For example, the activity type “play a video” might include a video stream input, events for playing and rewinding, and observable properties such as a playing timer, and the physical location of the player. Activity types may result from a community standardization effort, and end-users typically refer to lists of available types when creating team designs.

A **team design** describes a number of **activity sheets** and the communication paths between them. Activity sheets indicate the activity type and add the definition of output events with associated **triggering conditions** and **data payload**. Communication paths link output events to input events on different sheets. To promote ease-of-use and resilience to end-user mistakes, events may be freely connected: matching of fields in the payload is best effort, based on field name. Players should provide reasonable defaults to all fields.

The **trigger engine** runs on the players and evaluates the trigger conditions based on the input and player events. **Team editors** enable end users to edit team designs, including commands to deploy and retire a team.
Figure 1 illustrates an example of a TeC team[2]. The purpose of the deployed team is to monitor the perimeter of Anne's small farm. The fence is equipped with sensors that detect animals leaning against the wire, or breaking it. If either event happens, a call is sent to Anne's cell phone, streaming video captured from the fence so that the situation can be assessed. Anne may press key 5 on her phone, indicating that no further action is required from the system.

In summary, a TeC team is represented by a team design, which includes activity sheets. The activity sheets describe the roles of each team player on the team and the way that the players communicate with each other. End users create team designs via a team editor. They can deploy, modify, and retire TeC teams by interacting with the editor.

Essentially, a running TeC team is a decentralized distributed system. Besides addressing all of the issues faced by a distributed system, such as communication, synchronization, fault tolerance and so forth, Team Computing typically has its own unique challenges. These challenges include the following aspects:

- Because the end users will design, deploy and update their TeC teams as they wish, the TeC framework must be flexible.
- A TeC framework needs to be resilient, due to incomplete or inconsistent team designs provided by the end users.
- A TeC framework should be capable of handling mobility for the team members and should work gracefully in a dynamic environment.
- All of the team members in a TeC team will play their roles autonomously, and the overall behavior of any team is emergent.
- A team member typically has only very limited resources, such as CPU and power.

Even though the TeC model has been in existence for more than two years, no working system exists that fully realizes all of its features. A few academic systems have been attempted, which are discussed further in the Related Work section; however, they are not generic and lack some of the major components, such as trigger evaluation and self-healing.

Except for reusing an existing TeC editor, this project has implemented all of the TeC framework components: team communication, team player, trigger evaluation, and self-healing from scratch on the popular Android platform. Its design follows the principle of 'Separation of the Concerns'[4]. Therefore, it is highly modularized and very portable.

For example, if other people need a similar trigger evaluation feature in a different system, they can easily extract the trigger engine component and plug it into their code. Or if there is a new messaging middleware being developed, we can simply replace the messaging module in our TeC framework and the system will still work fine.
All of the components implemented by this project can work together and provide a complete TeC system, which allows various TeC teams to be running on Android devices. The project then evaluates the system based upon criteria such as network traffic and power consumption. It also compares two different approaches for TeC communication.

Finally, the reason this project is implemented on the Android platform is that Android is an open system with a large development community. It is not only the world's most widely used smart phone and tablet platform[5], but it also boasts applications on televisions, games consoles and other electronics[6].

In 2011, Google announced its Android@Home project[7], which focused on home automation. The capabilities in Android code for home automation have been included in the Android 4.2.2 update[8]. In addition, Google has developed its own wireless protocol and Android@Home framework that allows Android applications to discover, connect, and communicate with electrical appliances and devices in the home. Unfortunately, all of their work is still proprietary[7].

In summary, the contribution of this project is not only the implementation of a working TeC system, which demonstrates the TeC concepts, but also the establishment of a baseline for future projects by virtue of its clean design and modularized open source code. Hopefully, this project can be a showcase of team computing and will further promote the TeC model.
RELATED WORK

Ubiquitous Computing was first introduced by Dr. Mark Weiser around 1988 at Xerox Palo Alto Research Center[9]. In general, it means that people simultaneously engage some computational devices and systems without necessarily being aware that they are doing so. More formally, Ubiquitous Computing is defined as "machines that fit the human environment instead of forcing humans to enter theirs"[10]. Sometimes Ubiquitous Computing is also described as Pervasive Computing[11].

Distributed Pervasive Systems have been discussed in many research papers. In the paper titled “System Support for Pervasive Applications”[12], three requirements have been defined for a pervasive application:

1. Embrace contextual changes, which means that a device in a pervasive system must continuously be aware that its environment may change at any time;

2. Encourage ad hoc composition, which means that many devices in the systems will be used in very different ways by different users. As a result, it should be easy to configure the application running on a device, either by the user or through automated interposition; and

3. Recognize sharing as the default, meaning the devices in a pervasive system should be able to easily read, store, manage and share information. They should also be able to efficiently discover services and react accordingly.
In the book *Distributed Systems - Principles and Paradigms*, there is a section that discusses distributed pervasive systems such as home systems and sensor networks. Particularly, the location where the data should be processed and how the devices should cooperate are discussed in detail. For instance, should the device simply send its data to a centralized node, or should it process the data by itself[13]? In the TeC model for this project, the issue is addressed by the Trigger Engine component and the data is processed on each device.

Prior work has been done by Dr. Joao Sousa and the teams he led. Notably, the Team Computing foundations and operation semantics were presented in the paper: “Foundations of Team Computing: Enabling End Users to Assemble Software for Ubiquitous Computing”[3].

In the papers “TeC: end-user development of software systems for smart spaces”[14] and “Space-Aware TeC: End-User Development of Safety and Control Systems for Smart Spaces”[2], the TeC framework is elaborated and expanded. Some concrete examples were provided in those papers, as well.

In addition, the concepts of self-adaption and self-healing were introduced in the paper “Towards User Tailoring of Self-adaptation in Ubiquitous Computing”[15].

In terms of implementation of TeC, Daniel Keathley and Luan Pham et al. have implemented an Android based prototype to demonstrate how the team editor and players work together[16]. Frank Hodum has completed an Android location based ad-hoc networking sub-system[18]. Vasilios Tzeremes is working on an implementation for product line based TeC design meshing[19].
The ideas used and implemented in this project are attributable to all of the mentioned research papers. However, from engineering's perspective a full-fledged working system that demonstrates all of the TeC concepts is still nonexistent. The previously implemented TeC prototype is more like a proof of concept for only the interaction between editor and players. Frank Hodum's ad-hoc network component, unfortunately, only works for Android 2.x, which is now retiring. Vasilios Tzeremes's project focuses on the user interface and is still a work in progress. Meanwhile, this project is trying to address all of those issues and fill all the gaps.

This project has reused one component from the prior work by Vasilios Tzeremes specifically his team editor implementation[19]. For the other major TeC components, such as communication protocol, team player, trigger engine, and self-healing, this project has its own design and implementation. This project not only demonstrates a running TeC system but also open sourced all of its code. Hopefully it can lay a solid foundation for Team Computing and make future projects much easier.
USE CASES

In order to implement the project, the following critical use cases are identified. They are grouped in three functional domains: Team Management, Players Interaction in a Team and Self-healing.

Please note that only those use cases that are going to be implemented by this project are listed here. Some TeC use cases, such as creating team design (design meshing) and setting up an ad-hoc network, would be implemented by other teams in different projects.

The following actors are involved in the use cases for this project:

<table>
<thead>
<tr>
<th>Actor</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Editor</td>
<td>The Editor enables end users to edit team designs, issue commands to deploy team designs, cancel deployed team designs etc.</td>
</tr>
<tr>
<td>Team Manager</td>
<td>Based upon the commands from the Editor, the Team Manager is responsible for managing the team. Duties include discovering the players, deploying a team, canceling a team, etc.</td>
</tr>
<tr>
<td>Player</td>
<td>The Player interacts with the team, including receiving team briefings and input from other players, and sending output based on certain conditions.</td>
</tr>
</tbody>
</table>
Team Management

1) **Add Device to TeC Environment**

**Description:**

This use case describes how a physical device, such as a smart phone or a motion sensor, joins a TeC environment.
Actors:
Editor, Team Manager

Flow of events:

Basic flow

1. The Editor scans the barcode of the physical device;
2. The Editor creates a device record and saves the barcode as the initial key;
3. The Editor saves the initial key in its data store;
4. The Editor invokes the Team Manager and passes the device record to the Team Manager;
5. The Team Manager sends a message encrypted using the initial key to the admin multicast group. This message contains a randomly generated session key created by the Team Manager;
6. The device receives the session key message and tries to decrypt the message using the initial key. If the message makes sense, the device retrieves the session key for further communications;
7. The device is ready for discovery.

Alternate flow:
None

Pre-conditions:
The Editor and Team Manager are both running. The device has already started up, and the TeC application is running as well, but the device is not joined to the TeC environment, that is, it cannot communicate to anybody in the environment.

Post-conditions:
The device successfully joins the TeC environment and is ready for discovery.

2) **Discover Players**

**Description:**

This use case describes how the Team Manager discovers players.

**Actors:**

Team Manager, Player

**Flow of events:**

**Basic flow**

1. The Team Manager creates a discovery message;
2. The Team Manager sends the message to the admin multicast group;
3. The Player responds to the discovery message with a reply message;
4. The Team Manager receives the reply message;
5. The Team Manager records the player information from the reply message, which contains the player id and activity types that the Player supports.

**Alternate flow:**

1. The Team Manager creates a discovery message with a condition, which could be based upon location, activity types etc.
2. The Team Manager sends the message to the admin multicast group;
3. The Player receives the message and checks the condition;
4. The Player will respond only if the condition is true;
5. The flow resumes from basic flow step 4.

**Pre-conditions:**
The Team Manager does not have knowledge about the players.

**Post-conditions:**

The Team Manager acquires a list of all of the players responding to its discovery message. The Team Manager also receives all of the activity types that the discovered players are supporting.

3) **Deploy Team**

**Description:**

This use case describes how a Team Manager deploys a team based upon the command from the Editor.

**Actors:**

Editor, Team Manager

**Flow of events:**

**Basic flow**

1. The Editor issues the “deploy team” command to the Team Manager;
2. The Team Manager extracts the team design and all of the related activity sheets from the database;
3. The Team Manager tries to discover the relevant players, based upon the activity sheets (use case 2);
4. If the Team Manager cannot discover all of the needed players, it issues an error and returns to the Editor;
5. The Team Manager sends a briefing message to each player, which includes the activity sheet;
6. The Team Manager reports to the Editor that the team is deployed.

**Alternate flow:**
Pre-conditions:
None

Post-conditions:
The briefing is sent.

4) **Retire Team**

**Description:**
This use case describes how the Team Manager retires an already deployed team per the “retire” command from an Editor.

**Actors:**
Editor, Team Manager and Player

**Flow of events:**

**Basic flow**

1. The Editor issues a “retire” command;
2. The Team Manager creates a retiring message that includes the team id;
3. The Team Manager sends the message to the admin multicast group;
4. The Player receives the retiring message and stops operating for the team;
5. The Team Manager reports to the Editor that the team is retired.

**Alternate flow:**

None

Pre-conditions:
None

Post-conditions:
The retiring message is sent, and the Team Manager reports that the team is retired.

**Players Interaction in a Team**

![Diagram showing the flow of events](image)

**Figure 3 Use Cases for Players Working in a Team**

5) **Receive Briefing**

**Description:**

This use case describes how a Player receives a briefing.

**Actor:**

Player

**Flow of events:**

**Basic flow**

1. The Player receives a briefing message from the admin multicast group;
2. The Player extracts the activity sheet from the briefing message and records it;
3. The Player parses the triggers for the outputs and saves them in the trigger engine;
4. The Player spawns a new thread to listen at the team multicast group waiting for inputs.

**Alternate flow:**

1. If the Player discovers that the briefing is for the same team, it extracts the activity sheet from the briefing message and updates the existing activity sheet;
2. The Player parses the triggers for the outputs and updates them in the trigger engine;
3. If there is no thread listening at the team multicast group waiting for inputs for that team, the Player will spawn a new thread that does so.

**Pre-conditions:**
The Player is running.

**Post-conditions:**
The activity sheet is recorded/updated by the Player and the Player is waiting for inputs and is ready to process inputs and events.

6) **Receive Retiring**

**Description:**
This use case describes how a Player receives a retiring message.

**Actor:**
Player

**Flow of events:**

**Basic flow**

1. The Player receives a “retire” message from the admin multicast group and retrieves the team id from the message;
2. The Player checks if it is operating for the team. If it isn’t, it ignores the message, otherwise, it continues to the next step;

3. The Player terminates the working thread for the team and stops listening at the team multicast group;

4. The Player runs the cleanup function to remove the triggers from the trigger engine and performs some device specific cleanup, if necessary.

**Alternate flow:**

None

**Pre-conditions:**

The Player is operating for the team.

**Post-conditions:**

The Player stops operating for the team.

7) **Receive Input**

**Description:**

This use case describes how the Player receives an input and processes it. For each Player, based upon its capabilities and activities types, the input will trigger some specific native functions to perform some real work.

**Actor:**

Player

**Flow of events:**

**Basic flow**

1. The Player receives an input message from the team multicast group;

2. The Player invokes its input processing function;
3. The input processing function may invoke some native functions to perform some device specific work, and it puts the input into a queue.

Alternate flow:

None

Pre-conditions:
The Player is running, waiting for input.

Post-conditions:
The input has been received and processed by the Player. Some player-specific functions have been performed.

8) **Evaluate Trigger** *(See the Trigger Evaluation section for more technical details)*

Description:
This use case describes how the Player evaluates the trigger based upon the inputs and the events it receives.

Actor:
Player

Flow of events:

Basic flow

1. The Player receives an event or input;
2. The Player sets the property variable value based upon the event and puts it into a queue;
3. The trigger engine picks up the variable from the queue and evaluates it with the available triggers;
4. If a trigger becomes true due to the variable change, the output function will be invoked.
Alternate flow:
None

Pre-conditions:
The Player is running and waiting for an event.

Post-conditions:
The event has been received and processed by the Player. The output function may be invoked.

9) **Send Output**

Description:
This use case describes how a Player sends out an output when the output trigger becomes true.

Actor:
Player

Flow of events:

Basic flow

1. When a registered trigger becomes true, the trigger engine invokes the Player’s output function;

2. The Player creates an output message based upon the output payloads;

3. The Player sends the output message to the team multicast group.

Alternate flow:
None

Pre-conditions:
The Player is running without sending out the output message.

Post-conditions:
The trigger becomes true in the trigger engine for the Player and an output message is sent to the team.

Self-healing

10) Establish Bond

Description:

This use case describes how a bond is established between two Players.

Actors:

Healer, Player
Flow of events:

Basic flow

1. Player A determines that it is not in a bond relationship;
2. Player A creates a “Bind request” message and sends it to the admin multicast group;
3. A potential Healer receives the “Bind request” message and determines if it is available for the bond;
4. The Healer responds with a “Healer available” message to Player A;
5. Player A receives the “Healer available” message and sends the healer its activity sheets, that is, Player A briefs the Healer.
6. The Healer receives the briefing message and records the activity sheets;
7. Both the Healer and Player A record the bond relationship.

Alternate flow:

During any step before the bond relationship is set up, if a time-out occurs for either the Player or the Healer during the message exchange, no bond will be created. The procedure must be started over again if the Player still wants a bond relationship.

Pre-conditions:

No bond relationship between the Healer and the Player.

Post-conditions:

A bond relationship has been established between the Healer and the Player.

11) Monitor Player

Description:

This use case describes how after a bond has been established, the Healer monitors the other Player.
**Actors:**
Healer, Player

**Flow of events:**

**Basic flow**

1. The Healer periodically sends a heartbeat message to the Player it monitors;
2. The monitored Player receives the heartbeat message and responds with a reply message;
3. The Healer receives the reply message and updates the heartbeat status for the Player.

**Alternate flow:**

Alternate flow one:

1. If the monitored Player does not receive the heartbeat message for some time, it will conclude that the Healer is gone and will look for a new Healer (establish a new bond with a new Healer).

Alternate flow two:

1. If the Healer does not receive the reply message from the monitored Player for some time, it will conclude the Player is gone and record the status;
2. The Healer will send out a message in the admin multicast group to report that the monitored Player is gone;
3. The Healer will then try to start the “recover player” use case.

**Pre-conditions:**
Both the Healer and the monitored Player are running.

**Post-conditions:**
Either the Healer or the monitored Player has failed. The situation is recognized and being processed.
12) **Recover Player**

**Description:**

This use case describes how the Healer tries to recover when its player fails.

**Actors:**

Healer, Players

**Flow of events:**

**Basic flow**

1. The Healer discovers that its monitored Player is not responding;
2. The Healer examines the activity types based upon the Player’s activity sheets;
3. The Healer sends out discover messages for a new Player to replace the failed one;
4. If a new Player responds, the Healer will brief the new Player;
5. The Healer sends “bind message” to the new Player and records the status.

**Alternate flow:**

Alternate flow one:

1. After step 3, if no Player responds and the Healer itself is in a bond relationship, the Healer will update its own Healer with the failed Player’s briefing.
2. The Healer will periodically send out a discover message. If a Player responds at some point, resumes from step 4;
3. If the Healer fails before it finds a replacement, the Healer’s Healer (if there is one) will take over the discover process and keep searching for a suitable Player.

**Pre-conditions:**

Either the Healer or the Player has failed. The bond relationship is broken.
**Post-conditions:**

A new Healer or Player joins the team, and a new bond relationship is established.
SYSTEM ARCHITECTURE

For a TeC system, the key components include Players, Editors of team designs, Team Managers (TMs), which are present at each space, and Device Managers, which control the physical devices (see Figure 5).

![System Architecture Diagram]

Figure 5 System Architecture of a TeC Environment
While some editor functionalities are needed, this project mainly focuses on Team Managers, Players and how their interactions with each other in a TeC environment. Specifically,

- The Team Manager oversees the set of Players within a geographical area. It also contains other components that facilitate functions such as deploying, updating, and retiring a team;

- Players are computing-enabled devices that participate in a TeC system;

- Editors enable end users to edit team designs. They also display commands to deploy and retire a team.

- A Device Manager wraps the functionalities of the physical device. It also acts as a controller that manages different components on the player side;

- The communication protocols used in the project are mainly UDP multicast. However, if point-to-point communication is needed, such as stream a video, TCP/IP is also used.

This project will try to reuse other teams’ code for the editor functionalities. Therefore, some of the editor features will not be in the project scope.
DATA MODEL

The system will not accommodate a huge amount of data. However, it still needs a relational database as a persistent data store. The relational database serves two purposes:

- It is a data store for all of the type information needed when the user is working on the team design.
- It saves all of the team design data once the user completes the design using the editor. During the team deploying, updating, and retiring steps, the Team Manager will read the database and obtain all of the needed data.

The following diagram shows all of the tables and data fields the project is going to use. All of the relationships between the tables are also shown in the diagram.

Figure 6 Data Model for the Relational Database
DESIGN & IMPLEMENTATION

Overview

The development methodology used for this project is quite agile: not all of the requirements were determined up front, and almost every week during the school semesters I would meet with my advisor and discuss the design, project progress, technical difficulties, and so on. Project requirements and design decisions were often modified based upon the feedback I received. During the design phase, I also built three prototypes: one for the trigger engine; one for the backend running in a traditional PC environment; and the third, an Android based Proof of Concept system.

Network

All of the Players and Editors in a TeC environment need to communicate with each other via a network. Nowadays, Wi-Fi is a common and natural choice for TeC. Since the design only relies upon UDP multicast, the system will run fine as long as multicast is supported by the network. These are the choices for the project:

- **Wireless Access Point**

  Typically, a wireless access point is provided by a wireless router. Almost all modern wireless routers support broadcast/multicast. This alternative provides a convenient way for us to test the software and was used very often in the Proof of Concept project.

- **Wireless Ad-hoc Network**
A wireless ad-hoc network connects two or more devices without using a wireless access point[17]. The devices communicate directly when in range. This is the approach that was adopted in the first phase of the project when Frank Hodum’s Android ad-hoc network library was used[18]. Unfortunately, the library is not compatible with Android 4.0 and later versions.

➢ **Wi-Fi Direct**

Wi-Fi Direct is a standard that allows Wi-Fi devices to connect to each other without the need for a wireless access point[23]. Wi-Fi Direct essentially implements a “Software Enabled Access Point” ("Soft AP") into the network, thus eliminating any external devices. Since Android 2.3, Wi-Fi Direct has been supported by the OS, and it has been used in the implemented system.

➢ **Multicast in Mobile Network**

There have been some discussions about multicast in a mobile cellar network. At this moment, it is use more in the academia and is not useful for our implementation. It might become a possibility in the near future, given information technology is advancing so rapidly.

**TeC Message**

➢ **Message Format**

A TeC environment is essentially message driven. All of the Players and Editors send and receive messages from each other. Any TeC message has a message header and a
message payload. The message payload is either a simple text string or a piece of XML code (Figure 10 is an example).

A message header contains the message id, TeC team id, and other useful information such as the editor id and the receiving device id, if it is available.

➢ **Message Serialization**

If the body of the message is in XML format, it will be a serialization of a TeC class object. The framework *Simple XML*[24] is used for this purpose and to perform the serialization/de-serialization work.

For instance, the class ActivityType has the *Simple XML* annotations shown in Figure 7. When it is serialized, a correct XML object will be generated. This XML object can then be passed to other Players via the network, and be de-serialized back to the ActivityType object in a different address space.

```java
import org.simpleframework.xml.Attribute;
import org.simpleframework.xml.ElementList;
import org.simpleframework.xml.Root;

@Root(name = "activity-type")
public class ActivityType {
    @Attribute(required = false)
    private String name;
    @Attribute(required = false)
    private String short_description;
    @Attribute(required = false)
    private String long_description;

    @ElementList(name = "input-type", required = false, inline = true)
    private List<InputType> inputTypes;

    ......
}
```

*Figure 7 A "Simple XML" Example*
Security Considerations

➢ Wi-Fi Security

A typical Wi-Fi network would already have some sort of security mechanism[20]. For example, a company Wi-Fi network probably has WPA2-Enterprise implemented, which requires authentication to use it and has strong encryption. Moreover, Wi-Fi Direct has built-in security: WPA2 is bundled into this wireless standard.

➢ TeC Security

In this TeC implementation, additional security has been built into the system. We require different keys in the communication, and thus encrypted the message payload accordingly.

At the beginning, any device that wants to join a TeC team has to be scanned by a super node in the environment, such as a team Editor. The scanning result includes a device key, which comes with the device. An encrypted session key will be sent to the new device, and then it can start to communicate with the Editor. Once a TeC team is deployed, all of the players in the team will form a random multicast group and start using a team key to communicate with each other.

Editor

The following table and figure show all of the major classes and their relationships for the team manager sub-system.
Table 2 Editor Classes

<table>
<thead>
<tr>
<th>Class Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>AndroidTecEditor</td>
<td>This class implements a simplified editor using command line interface. It provides limited editor functionalities.</td>
</tr>
<tr>
<td>TeCDBAdhoc</td>
<td>This class talks to the databases and implements the DB functionalities.</td>
</tr>
<tr>
<td>TeamManager</td>
<td>This class works as the Team Manager in TeC.</td>
</tr>
<tr>
<td>TeamDesign</td>
<td>This class contains all the data and methods necessary for team designs.</td>
</tr>
<tr>
<td>ActivitySheet</td>
<td>This class contains all of the data and methods necessary for activity sheets.</td>
</tr>
<tr>
<td>ActivityType</td>
<td>This class contains all of the data and methods necessary for activity types.</td>
</tr>
<tr>
<td>InputType</td>
<td>This class contains all of the data and methods necessary for the input types.</td>
</tr>
<tr>
<td>Input</td>
<td>This class contains all of the data and methods necessary for the player inputs.</td>
</tr>
<tr>
<td>Output</td>
<td>This class contains all of the data and methods necessary for the player outputs.</td>
</tr>
<tr>
<td>AdminMessenger</td>
<td>This class allows the Editor to communicate in an admin multicast group.</td>
</tr>
<tr>
<td>UdpAdminPackage</td>
<td>This class works with the Wi-Fi network and implements UDP multicast functionalities.</td>
</tr>
</tbody>
</table>
Player

The following table and figure show all of the major classes and their relationships for the player sub-system.
<table>
<thead>
<tr>
<th>Class Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Player</td>
<td>This class is the parent class for all of the concrete players, such as smart phones or sensors. It implements all of the common methods for all of the Players.</td>
</tr>
<tr>
<td>DeviceManager</td>
<td>This class wraps the physical device driver and initializes all of the Players.</td>
</tr>
<tr>
<td>SmartPhone</td>
<td>This class simulates a smart phone in the project.</td>
</tr>
<tr>
<td>MotionDector</td>
<td>This class simulates a motion detector in the project.</td>
</tr>
<tr>
<td>Camera</td>
<td>This class simulates a camera in the project.</td>
</tr>
<tr>
<td>SmokeDetector</td>
<td>This class simulates a smoke detector in the project.</td>
</tr>
<tr>
<td>Camcorder</td>
<td>This class simulates a camcorder in the project.</td>
</tr>
<tr>
<td>TriggerEngine</td>
<td>This class is the proxy class. It works between the trigger engine sub-system and the player sub-system.</td>
</tr>
<tr>
<td>ActivitySheet</td>
<td>This class contains all of the data and methods necessary for activity sheets.</td>
</tr>
<tr>
<td>ActivityType</td>
<td>This class contains all of the data and methods necessary for activity types.</td>
</tr>
<tr>
<td>TeamMessenger</td>
<td>This class allows the Players to communicate in a team multicast group.</td>
</tr>
<tr>
<td>UdpCommonsPackage</td>
<td>This class works with the Wi-Fi network and implements UDP multicast functionalities in the team multicast group.</td>
</tr>
<tr>
<td>AdminMessenger</td>
<td>This class allows the Editor to communicate in an admin multicast group.</td>
</tr>
<tr>
<td>UdpAdminPackage</td>
<td>This class works with the Wi-Fi network and implements UDP multicast functionalities in the admin multicast group.</td>
</tr>
</tbody>
</table>
Trigger Evaluation

A trigger engine is designed to evaluate the triggers for output. A trigger could be a simple expression such as "lean | break" as shown in Figure 10.

```xml
<activity-sheet activity="monitor fence">
  <out-evt>
    <evt name="alert" trigger="lean | break">
      <att name="dial" value="703 111 1234"/>
      <att name="msg" value="There's a possible break in the perimeter."/>
      <target in-evt="call" ip="phone ip" port="***" key="******"/>
      <target in-evt="on" ip="film ip" port="***" key="******"/>
    </evt>
  </out-evt>
</activity-sheet>
```
Alternately, it could be a complex expression contains functions such as PostP(), Rest() and Toggle().

For example the expression "Detect_Smoke & PostP((Kitchen_Temperature > 500 | Bedroom_Temperature > 400), 30)" will be true if all of the following are true:

1. Smoke is detected;
2. The temperature of the kitchen is greater than 500F or the temperature of the bedroom is greater than 400F;
3. After 30 seconds, #2 is still true.

The following table defines how the trigger can be formed.

<table>
<thead>
<tr>
<th>$\varphi$ ::= $p$</th>
<th>$\varphi$</th>
<th>$\neg \varphi$</th>
<th>$\varphi_1 \land \varphi_2$</th>
<th>$\varphi_1 \lor \varphi_2$</th>
<th>PostP($\varphi$, $t$)</th>
<th>Rest($\varphi$, $t$)</th>
<th>Toggle($\varphi_1$, $\varphi_2$)</th>
</tr>
</thead>
<tbody>
<tr>
<td>a predicate</td>
<td>$\delta$</td>
<td>logical negation</td>
<td>logical conjunction</td>
<td>logical disjunction</td>
<td>once $\varphi$ becomes true, $\textit{postpone}$ becoming true by time $t$</td>
<td>becomes true when $\varphi$ becomes true, and then $\textit{rests}$ for time $t$ before examining $\varphi$ again</td>
<td>becomes true when $\varphi_1$ becomes true, false when $\varphi_2$ becomes true, true again with $\varphi_1$ and so forth</td>
</tr>
</tbody>
</table>

Figure 11 Form of triggering conditions[3]
Since the trigger could be very complex, the open source package ANTLR (ANother Tool for Language Recognition)[25] is used to parse the trigger expression and evaluate the trigger once the trigger variables have changed.

The following table and figure show all of the major classes and their relationships for the trigger engine sub-system.

<table>
<thead>
<tr>
<th>Class Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>TriggerEngine</td>
<td>This class is the proxy class. It works between the trigger engine sub-system and the player sub-system.</td>
</tr>
<tr>
<td>Processor</td>
<td>This an inner class for the TriggerEngine. It receives events and calls the evaluator to evaluate the trigger. Once the trigger becomes true, it invokes the registered callback.</td>
</tr>
<tr>
<td>Evaluator</td>
<td>This class evaluates the trigger expression by walking through the Abstract Syntax Tree.</td>
</tr>
<tr>
<td>TriggerAST</td>
<td>This class represents the Abstract Syntax Tree of the trigger expression.</td>
</tr>
<tr>
<td>ITriggerEvent</td>
<td>This is an interface for the user class to register a callback, so that when the trigger becomes true, the callback will be invoked.</td>
</tr>
<tr>
<td>TriggerXML</td>
<td>This class converts the trigger to AST or AST to XML.</td>
</tr>
<tr>
<td>TriggerLexer</td>
<td>This class extends the ANTLR Lexer class and serves as a lexer for the trigger.</td>
</tr>
<tr>
<td>TriggerParser</td>
<td>This class extends the ANTLR parser class and serves as a parser for the trigger.</td>
</tr>
<tr>
<td>TriggerNode</td>
<td>This class represents a node of the AST.</td>
</tr>
</tbody>
</table>
Use Case Implementation

- Scan Device Data:

One assumption is for each device it has a unique **Quick Response Code** (QR Code)[21]. The device QR code encodes the device information such as device id, unique device
key, and a short description. The following table shows an example of the QR code for a smart phone:

![QR Code Example](image)

<table>
<thead>
<tr>
<th>QR Code</th>
<th>Device Data</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Smart Phone I:phone_1:12345:Living room:it is a phone</td>
</tr>
<tr>
<td></td>
<td>Name: Smart Phone I</td>
</tr>
<tr>
<td></td>
<td>ID: phone_1</td>
</tr>
<tr>
<td></td>
<td>Key: 12345</td>
</tr>
<tr>
<td></td>
<td>Location: Living room</td>
</tr>
<tr>
<td></td>
<td>Description: it is a phone</td>
</tr>
</tbody>
</table>

A third-party library, ZXing, which is an open-source, multi-format 1D/2D barcode image-processing library implemented in Java[22], is used in the system to scan the QR Code and obtain the result of the scan. ZXing utilizes the built-in camera on the Android device to scan and decode barcodes without communicating with a server. The following code snippets illustrate how to invoke ZXing and obtain the result from a callback.

```java
public void addNewDevice(View view) {
    Intent intent = new Intent("com.google.zxing.client.android.SCAN");
    // Setting other parameters
    ......
    intent.putExtra("SCAN_MODE", "QR_CODE_MODE");
    startActivityForResult(intent, 0);
}
```

Figure 13 Invoke ZXing to Scan QR Code
Obtain Scan Result from ZXing

```java
public void onActivityResult(int requestCode, int resultCode, Intent intent) {
    if (requestCode == 0) {
        if (resultCode == RESULT_OK) {
            String contents = intent.getStringExtra("SCAN_RESULT");
            String values[] = contents.split(":");
            if (values.length < 4) {
                Toast.makeText(this,
                           "Oops, sorry the barcode seems not correct.",
                           Toast.LENGTH_SHORT).show();
            } else {
                DeviceRecord rec = new DeviceRecord();
                rec.setName(values[0]);
                rec.setId(values[1]);
                rec.setInitKey(values[2]);
                rec.setLocation(values[3]);
                if (values.length > 4)
                    rec.setDescription(values[4]);

                editor.saveDevice(rec);
                dlist.add(rec.getName() + rec.getId() + rec.getLocation());
                adapter.notifyDataSetChanged();
            }
        } else if (resultCode == RESULT_CANCELED) {
            Toast.makeText(this,
                           "Oops, sorry can't scan barcode for some reason.",
                           Toast.LENGTH_SHORT).show();
        }
    }
}
```

Team Management:

1) **Add Device to TeC Environment**

The following sequence diagram illustrates how the Editor interacts with a device and makes the device ready to join a TeC team.
2) **Discover Players**

The following sequence diagram illustrates how the Team Manager discovers Players.

![Sequence Diagram](image)

**Figure 16 Discover Players Sequence Diagram**

3) **Deploy Team**
The following sequence diagram illustrates how the Team Manager deploys a team based upon the command from the Editor.

![Deploy Team Sequence Diagram](image)

**Figure 17 Deploy Team Sequence Diagram**

4) **Retire Team**

The following sequence diagram illustrates how the Team Manager retires an already deployed team per the “retire” command from an Editor.
Players working in a team

5) **Receive Briefing**

The following sequence diagram illustrates how a Player receives a briefing.
6) **Receive Retiring**

The following sequence diagram illustrates how a player receives a retiring message.
7) **Receive Input**

The following sequence diagram illustrates how the Player receives an input and processes it. Each Player, based upon its capabilities and activities types, will invoke some specific native functions to perform some real work.

![Sequence Diagram](image)

Figure 21 Receive Input Sequence Diagram

8) **Evaluate Trigger**

The following sequence diagram illustrates how the Player evaluates the trigger based upon the inputs and the events it receives.
9) **Send Output**

The following sequence diagram illustrates how a Player transmits an output when the output trigger becomes true.

---

Figure 22 Evaluate Trigger Sequence Diagram
**Bself-Healing**

**Overview**

In Bond self-healing, *B*self-healing for short, each node $p$ is assigned a healer: another node who (a) receives enough information to recover $p$'s activity should $p$ fail; and (b) is responsible for monitoring $p$. The two nodes thus have a healing bond[15]. The following workflow describes the *B*self-healing implementation in the system.

**Healing Workflow**

1: After the team is initially deployed and the team players start running, each of the Players will search for a Healer by sending out a multicast message SELF_HEALING_REQUEST to its team. For instance, the following diagram shows a team with three players A, B, and C, multicasting the message.
2: When a Player receives the SELF_HEALING_REQUEST message, it may reply with a SELF_HEALING_REPLY message, which indicates that the player is willing to be the Healer for the requesting Player. The following diagram shows an example:
There is one caveat though. Not all Players are willing or able to work as a Healer possibly due to resource constraints. Player C in the above diagram is such an example.

3: When a Player receives the SELF_HEALING_REPLY messages from its peers, it selects only the first one and sends its briefing to the sender. The rest of the SELF_HEALING_REPLY messages will be ignored. After sending the SELF_HEALING_BRIEFING message, which includes the payload in the Player's briefing, the Player will mark the target player as its Healer. After the target player receives the message, it will add the sender to its "Monitor Player" list and save the briefing. Figure 26 provides an example:
Note: In case one Player, say P1, marks another player, P2, as its healer, but P2 never receives the briefing or somehow is not aware of being the Healer of P1, there could be a problem. This situation will be treated similarly to losing a player, which will be further explained in step 5.

4: After a Player finds a Healer, it starts monitoring the Healer by periodically sending the Healer a SELF_HEALING_PING_HEALER message. Once the Healer receives this message, it will send back a reply message. Similarly, a Healer will be periodically sending SELF_HEALING_PING_PLAYER messages to all the Players in its "Monitor Player" list. All of the Players that receive this message will reply as well.

After all of the Players find their Healers and start monitoring each other, the team is considered to be self-healing; bonding is complete. Figure 27 illustrates the process.
5: In the case of a team losing a Player, any PING messages to that Player will not be answered. For example, if Player B is gone from the following team, Player A and C will not receive any reply messages from B.
6: Since the PING messages are not answered, the Players that send those messages can conclude that the targeted player of those messages is not part of the team anymore. Therefore, they will take actions to search for a replacement and re-bond.

For instance, after Player B leaves the team in the above example, Player A and C will start sending SELF_HEALING_REQUEST messages again, since B used to be the Healer of A and C. Player A will also broadcast the SEARCH_PLAYER message in the Admin multicast group in order to find a replacement Player for B because A used to be the Healer of B.

The SEARCH_PLAYER message includes the activity types of Player B, so if a match is found by a potential player based upon the activity types, the potential player can reply to the message from A requesting to join the team. The following figure shows an example.
7: During the process of searching for a replacement for the Player that left the team, new bonding relationships probably will be established among the other Players. After a potential player appears, the team can recruit it by sending it the team session key and the briefing.

For instance, in the example we are using, when player B’ receives the SEARCHPLAYER message, it will send out a SEARCH_REPLY message, since it has the same activity types that Player B had. Now the tricky part becomes how to deliver the team session key to B’. Considering two scenarios:

1) If the team Editor is still available and recognizes player B’, it will send the session key to B’ by encrypting the session key using the device key of B’. Therefore, B’ can decrypt the message and obtain the session key;

2) If any other Players knows the device key of B’ (either by scanning it or from already having it in the cache), this other Player can also send the session key to B’ by encrypting the session key using the device key of B’. Figure 30 illustrates the process.
8: Once the new player obtains the team session key, the Healer will send the briefing to it using the HEALER_BRIEFING message and will update the "Monitor Player" list. An example is shown in Figure 31.
9: After the new Player becomes briefed, it will examine the briefing message it received. If a stale variable is found, it will send a SELF_HEALING_REQ_UPDATE message to all the other Players in the team. This message will have the old value for the stale variable. If a new value is available, it will be included as well. In this case, the other Players can update the variable based upon the old and new values.

If a new value is not available, once the other Players receive the SELF_HEALING_REQ_UPDATE message, they will check to determine if they have the new value and if they do, they will reply using the SELF_HEALING_UPDATE message. Thus, the newly joined Player can update its stale variables. The process is illustrates in Figure 32.

Figure 32 Self-healing 9: Update Stale Variables
Note: Currently only the variable that has a Player's IP address needs to be updated. Typically, the IP address is used for streaming media files between two Players.

10: After all the above work, the players that do not have a Healer will try to find one. Once all the Players have Healers, the team goes back to a stable bonding situation, such as in the following example:

![Figure 33 Self-healing 10: Bonding Complete Again](image-url)
ALTERNATIVE IMPLEMENTATION

For the system designed and described in the previous chapters, we have essentially implemented our own messaging subsystem and all of the related protocols. That gives us a lot of flexibility and makes the system efficient and lightweight. The messaging subsystem itself is based upon multicast, which is supported very well by almost all modern Android devices. However, like the most common multicast, the default one in the Android world is still UDP-based and by its nature, UDP is not reliable. Unfortunately, it is surprisingly tricky to implement reliable multicasting[27]. Fortunately, there are existing high-quality libraries such as JGroups to support multicast messaging reliably. In order to compare it with our ad-hoc multicast messaging subsystem, an alternative TeC system was also built based upon JGroups. The comparison is described in the following sections.

Introduction of JGroups

According to its website[28], “JGroups is a toolkit for reliable multicast communication. It can be used to create groups of processes whose members can send messages to each other”. The main features of JGroups follow:

- Group creation and deletion. Group members can be spread across LANs or WANs;
- Joining and leaving of groups;
• Membership detection and notification about joined/left/crashed members;
• Detection and removal of crashed members;
• Sending and receiving of member-to-group messages (point-to-multipoint); and
• Sending and receiving of member-to-member messages (point-to-point).

Based upon the feature list, JGroups can help us in two main areas. First of all, it provides a reliable multicast mechanism for the communications among all the Players and Editors. Secondly, it helps to maintain a global view for the teams and dramatically simplifies the self-healing process.

However, the advantages of JGroups come with costs: additional overhead to ensure that the multicast messages are delivered reliably and in order, plus “chattiness” to maintain a global view of all of the participants. These costs can be translated into more computing and power consumption, which are probably critical to those small or embedded Android devices.

On the other hand, under some circumstances, such as in a controlled environment with adequate devices, or where reliable communication is more important than “best effort delivery”, it is still worthwhile to have an alternative such as JGroups. One possibility is to include a switch parameter in the future system, which allows the users to choose between a more reliable system that consumes more resources and a “best effort” system, which utilizes fewer resources.

**Using JGroups**

1. **Setup JGroups**
Before starting to use JGroups, we must decide what parameters we need to set for JGroups, since it is very flexible and allows customization for various needs. There are two ways that customization can be performed: using a property file or programmatically when the system is running.

Since the TeC admin multicast group should be well known by all of the Players and Editors, we decide to customize JGroups for the admin group via a property file and ship the property file with the software. The following is the property file we are using for the TeC admin multicast group. The parameters such as IP, Port, and Timeout values are the most important ones we should specify.
Figure 34 XML Property File for JGroups
For team Players we choose to programmatically setup JGroups, which gives us the flexibility to separate different team multicast groups based upon IP or ports, for instance:

```java
stack.addProtocol(
    new UDP().setValue("ip_mcast", false).setValue("mcast_port", mh.getTeamPort()))
    .addProtocol(
      new MFING()
        .setValue("receive_on_all_interfaces", true)
        .setValue("send_on_all_interfaces", true)
        .setValue("mcast_addr", InetAddress.getByName("230.0.0.2"))
        .setValue("mcast_port", mh.getTeamPort())
    .addProtocol(new MERGE2())
    .addProtocol(new VERIFY_SUSPECT())
    .addProtocol(
      new NAKACK().setWoluse("mcast_xmit", false)
        .setValue("xmit_stagger_timeout", 200)
        .setValue("discard_delivered_msgs", true))
    .addProtocol(new UNICAST2()).addProtocol(new STABLE())
    .addProtocol(new GMS());
stack.init();
tmchannel.setName("player-" + mh.getTeamid() + "-" + Config.uuid);
tmchannel.setReceiver(this);
tmchannel.connect(mh.getTeamName());
tmchannel.setDiscardOwnMessages(true);
```

Figure 35 Setup JGroups Programmatically

2. Send and Receive Messages

After setup, a JGroups channel needs to be created in order to send and receive messages. Team players join a multicast group by connecting the channel to a group address, and they leave it by disconnecting. Messages sent over the channel are received by all group members that are connected to the same group\[28]\.

The following code shows how a TeC member creates a JGroups channel in an AdminMessenger. With a JGroups channel, sending and receiving messages is straightforward. Basically, the JGroups channel replaces the UDP package classes in our
original design. Since our system follows the principle of Object Oriented Programming, the rest of the system does not need to be changed at all.

3. Self-healing with JGroups

Self-healing with JGroups is quite different from self-healing with the original design. The reason is obvious: with JGroups we now maintain a global view for all of the members in the multicast group. A callback \texttt{viewAccepted()} is provided by JGroups, which will be invoked when the group membership changes either by having a new member joined or by losing a team member.

Accordingly, two methods \texttt{onTeamSizeDown()} and \texttt{onTeamSizeUp()} have been implemented and will be invoked in the callback. When the team size increases, we know

```java
public AdminMessenger(IAdminMessengerListener mlistener, InputStream configInputStrem) throws Exception {
    ...
    // setup JGroups channel
    amchannel = new JChannel(configInputStrem);
    amchannel.setName(Config.uuid);
    amchannel.setReceiver(this);
    amchannel.connect(TecConstants.ADMIN_CLUSTER);
    amchannel.setDiscardOwnMessages(true);
    ...
}

public void writeMessage(String key, TeamMessage msg) throws Exception {
    String msg = MessagePacket.getMessageString(key, msg);
    Message m = new Message(null, null, msg);
    try {
        amchannel.send(m);
    } catch (Exception e) {
        e.printStackTrace();
    }
}

@Override
public void receive(Message msg) {
    receiveMessage(msg.getSrc(), new MessagePacket(msg.getObject().toString()));
}
```

**Figure 36 Create Channel and Send/Receive Messages with JGroups**
that new team members joined the team, and we can try to setup the bonding relationship for those new members. When the team size decreases, we know that the team lost members either voluntarily or involuntarily. At that time, we can still utilize our self-healing protocols to heal the team. Figure 37 is the code snippet for viewAccepted().

```java
@Override
public void viewAccepted(View new_view) {
    Log.d(Config.LOG_TAG, "** view: " + new_view);
    int newSize = new_view.size();
    if (teamSize > newSize) {  // team shrinks.
        mlistener.onTeamSizeDown(new_view);
    } else if (teamSize < newSize) {  // team size increased.
        mlistener.onTeamSizeUp(new_view);
    }
    teamSize = new_view.size();
}
```

Figure 37 JGroups Callback Used for Self-healing

**Summary of JGroups**

JGroups is an easy-to-use toolkit that provides two major advantages for a TeC system. One is that a multicast messaging system delivers messages reliably and in order. The other is that it maintains a global view for the team members to simplify the self-healing process. However, the advantages come with the cost of the communication overhead, which might be unbearable for small or embedded Android devices.
SYSTEM EVALUATION

Evaluation Criteria

Once the development is complete and the system is successfully running on multiple Android devices as expected, system testing and evaluation is conducted. The goal of system testing is to make sure that the system runs correctly with all of the features we designed. Due to the unique constraints of TeC, the evaluation criteria mainly focus on the following system attributes:

- Communication Efficiency
- Resource Consumption

Evaluation Configuration

There are many different Android devices on the market right now. However, in order to have comparable evaluation results and simplify the experiments, Google Nexus 7 tablets are used for the evaluation[29]. For the wireless network, Wi-Fi Direct is chosen since it is supported by Nexus 7 and does not need an external wireless access point.

Since our design separates the physical characteristics of the team Players from the TeC functionalities, any two team Players will behave identically if they are not engaged in their unique device activities. Actually, if there is no interesting event for a team Player most of the time it will just listen to the network and do nothing else.
For example, as shown in Figure 38, in a small team three Players are working together. Nothing happens if the motion sensor does not detect anything. However, if motion is detected by the motion sensor, it will send a “motion detect” event to the video camera. The video camera will start to record video and send a “video start” event to the smart phone. Then the smart phone can connect to the video camera to start video streaming.

For the measurements taken for our evaluation, usually a small team like this is deployed

**Communication Efficiency**

Essentially, a TeC team is also a message driven distributed system. It will be very beneficial to know the communication cost, especially considering that many team Players might be running on a single limited network. Less network traffic also helps the scalability for a TeC system.

For the ad-hoc messaging system, there are two running threads facilitating the network communication: one is a message writer sending messages to a multicast group, and the other is a message listener attached to the same multicast group. The messaging system is very efficient. For example, the discovery process only involves two messages between
the Editor and the Player: discovery message and reply message. In addition, there will only be two UDP packets transferred on the network as well.

Similarly, during the team deployment, after the discovery process, only one briefing message will be sent to each Player. When the team is in operating mode, each Player will only transmit a message when an event occurs. The only component that continuously transmits messages every few seconds is the player self-healing thread. So overall, the network traffic is very low.

For the JGroups alternative, we have the same kind of TeC messages in the system. However, JGroups introduces some communication overhead. For instance, for the team Player when we setup JGroups, we programmatically add six JGroups protocols, and each JGroups protocol is associated to a running thread. Protocols such as MPing (to determine if members are still alive) and GMS (to maintain group membership) also will be continuously transmitting messages every few seconds. In order to deliver messages reliably, JGroups members will also send acknowledgement messages for each message received, which theoretically adds another 50% to overhead. Overall at least four threads need to be added and the network traffic roughly doubles for each TeC member.

Note: Technically a TeC member in a particular team will receive all of the multicast messages in the multicast group. However, those unrelated messages will be filtered out by the receiver based upon the message header, so they will not be processed at all.

**Resource Consumption**

Since a typical TeC system will be deployed on many embedded or mobile Android devices, the resource consumption is critical, because those devices usually run on
batteries and do not have a lot of computing power. In order to have a better idea of the resource consumption, some measurements have been made for a few typical TeC deployments. The results are listed in the following sections.

1. **Power Consumption**

The power consumption for a team Editor typically is negligible since it does not play an active role once a team is deployed. Therefore, the power consumption measurement has been done on the Player side with Bself-healing enabled.

For a team Player running with the ad-hoc multicast messaging system, during an eight-hour period the power consumption is between 20% to 25% as shown below:

![Battery Consumption Graph](image)

*Figure 39 Power Consumption for a TeC Player*

For a team player running with the JGroups multicast messaging system, during an eight-hour period the power consumption is about 30%, as shown below:
By comparing those two different approaches, we can see there is about a 5% difference in the power consumption, probably due to the overhead of JGroups.

2. CPU and Memory Consumption

For a team Editor with the ad-hoc multicast messaging system, most of the time the CPU usage is negligible. Its memory consumption is about 11 MB (Proportional Set Size) when there are 13 threads running as shown below.

Figure 40 Power Consumption for a TeC Player Using JGroups
For a team Editor using JGroups multicast, most of the time the CPU usage is also negligible. Its memory consumption is about 14 MB (Proportional Set Size) when there are 24 threads running, as shown below:
For a team Player using the ad-hoc multicast messaging, most of the time the CPU usage percentage is about 1%. Its memory consumption is about 7 MB (Proportional Set Size) when there are 17 threads running, as shown below:

![Figure 43 CPU & Memory for a TeC Player](image)

For a team player using JGroups multicast, most of the time the CPU usage percentage is also about 1%. Its memory consumption is about 10 MB (Proportional Set Size) when there are 43 threads running, as shown below:
By comparing those two different approaches, we can see that there are some significant differences and again that is probably due to the overhead caused by JGroups.
SUMMARY

TeC is a very interesting area in that all of the elements work autonomously, and there is no central control. It has some unique challenges in terms of cooperation, security, self-adaption, and self-healing. This project tries to implement the major functionalities of the TeC framework on the popular Android platform. The project contains more than 13,000 line of Java code[26] and works for many Team Computing examples. The project establishes a good foundation for more complex TeC projects, and hopefully the TeC concept will be a success and make a great impact on people’s lives.
REFERENCES


CURRICULUM VITAE

Xiang Shen received his Bachelor of Engineering from the University of Science & Technology Beijing in 1997 and his Master of Science from the Maharishi University of Management in 2003. He was employed as a software engineer in the industry for more than ten years. He has been a solutions architect since 2011.