Collaborative Web-Based Mapping of Real-Time Sensor Data

Cristian Gadea

Thesis submitted to the
Faculty of Graduate and Postdoctoral Studies
In partial fulfillment of the requirements
For the MASc degree in Computer Science

Ottawa-Carleton Institute for Computer Science
School of Information Technology and Engineering
University of Ottawa
February 2011

© Cristian Gadea, Ottawa, Canada, 2011

UNIVERSITY OF OTTAWA

Abstract

Faculty of Graduate and Postdoctoral Studies
Ottawa-Carleton Institute for Computer Science

Master's Degree

by Cristian Gadea

The distribution of real-time GIS (Geographic Information System) data among users is now more important than ever as it becomes increasingly affordable and important for scientific and government agencies to monitor environmental phenomena in real-time. A growing number of sensor networks are being deployed all over the world, but there is a lack of solutions for their effective monitoring. Increasingly, GIS users need access to real-time sensor data from a variety of sources, and the data must be represented in a visually-pleasing way and be easily accessible. In addition, users need to be able to collaborate with each other to share and discuss specific sensor data. The real-time acquisition, analysis, and sharing of sensor data from a large variety of heterogeneous sensor sources is currently difficult due to the lack of a standard architecture to properly represent the dynamic properties of the data and make it readily accessible for collaboration between users. This thesis will present a JEE-based publisher/subscriber architecture that allows real-time sensor data to be displayed collaboratively on the web, requiring users to have nothing more than a web browser and Internet connectivity to gain access to that data. The proposed architecture is evaluated by showing how an AJAX-based and a Flash-based web application are able to represent the real-time sensor data within novel collaborative environments. By using the latest web-based technology and relevant open standards, this thesis shows how map data and GIS data can be made more accessible, more collaborative and generally more useful.

Acknowledgements

This thesis would not have been possible were it not for the guidance and encouragement of Dr. Dan Ionescu. His input and extensive experience was invaluable in the realization of this work.

I would also like to acknowledge the generous assistance of the other members of the University of Ottawa's Network Computing and Control Technologies Research Laboratory. Their depth of knowledge and devotion to technical research created a positive working atmosphere for me that was apparent from day one of the Master's program. Bogdan Solomon, Bogdan Ionescu, Robin Tropper and Rabih Dagher all contributed significantly to the projects mentioned in this thesis.

Finally, I would like to thank my parents and my brother for their continuous love and support throughout this journey.

Contents

Abstract i

Acknowledgements ii

List of Figures v

List of Tables vii

Abbreviations viii

1 Introduction 1
- 1.1 Motivation and Research Objectives . . . 1
- 1.2 Thesis Contributions . . . 6
- 1.3 Thesis Organization . . . 7

2 Background 9
- 2.1 Web Mapping Technology Overview . . . 9
- 2.2 The Need for an Architecture . . . 14
- 2.3 Related Work . . . 17
  - 2.3.1 Web Mapping Standards . . . 17
    - 2.3.1.1 RTCM Standards . . . 17
    - 2.3.1.2 OGC Standards . . . 19
  - 2.3.2 Real-Time Web Architectures . . . 22
    - 2.3.2.1 Basic Publisher/Subscriber Architecture . . . 22
    - 2.3.2.2 XMPP-Based Architecture . . . 24
    - 2.3.2.3 JMS-Based Architecture . . . 27
  - 2.3.3 Collaborative Platforms . . . 29
    - 2.3.3.1 Google Wave . . . 29
    - 2.3.3.2 Adobe Tour Tracker . . . 31

3 High Level Design 34
- 3.1 Architecture Requirements . . . 35
- 3.2 Component Design . . . 36

3.3 Component Descriptions 37

4 Low Level Design **47**
4.1 A Real-Time Platform for Multi-Domain Collaboration 48
4.1.1 Application Server 48
4.1.2 Collaborative Web Client Platform 52
4.1.3 GIS Web Application 56
4.2 A Real-Time Platform for Collaborative Multimedia 60
4.2.1 Application Server 61
4.2.2 Collaborative Web Client Platform 65
4.2.3 GIS Web Application 70
4.3 Shared Components 73
4.3.1 Subscriber 73
4.3.2 Publisher/Subscriber Server 76
4.3.3 Smart Publisher 77
4.3.4 Sensor Server & Spatial Database 80
4.3.5 Sensor Feeder 81
4.3.6 Sensor 83
4.3.7 Sensor Network 84
4.3.8 Sensor Registry 84

5 Results **86**
5.1 UC-IC Results 86
5.1.1 Basic Collaboration 87
5.1.2 Basic GIS Collaboration 88
5.1.3 General GIS Data 90
5.2 Watch Together Results 92
5.2.1 Facebook Deployment 92
5.2.2 Internal Test Deployment 93
5.3 Real-Time GIS Data 94
5.3.1 Public Sensor Data 95
5.3.2 Vehicle GPS Data 96
5.3.3 Aircraft Data 98

6 Conclusion **101**
6.1 Concepts Addressed in this Thesis 101
6.2 Contributions of this Thesis 102
6.3 Future Research 104

Bibliography **105**

List of Figures

1.1 LINET Sensor Network. 3

2.1 Typical Three-Tier Architecture. 10
2.2 Classic MapQuest Interface. 11
2.3 Classic vs. AJAX Application Model. 12
2.4 Multiple Windows and Menus in ESRI's ArcGIS. 16
2.5 Observations & Measurements Model. 20
2.6 Observer Pattern for Sensor Data. 23
2.7 XMPP-Based Architecture for SAS. 25
2.8 Publisher/Subscriber Messaging Using JMS. 28
2.9 JMS-Based Architecture for SOS Data. 28
2.10 Collaborative Google Maps in Google Wave. 30
2.11 Real-Time Data in Adobe Tour Tracker 2009. 32

3.1 GIS Web Application Use Case Diagram. 39
3.2 Collaborative Web Client Platform Use Case Diagram. 41
3.3 High Level Publisher/Subscriber Message Sequence Chart. 43
3.4 High Level Registry Message Sequence Chart. 45
3.5 Real-Time Web-Based GIS Architecture. 46

4.1 UC-IC System View. 49
4.2 UC-IC Subsystem View. 49
4.3 UC-IC Server-Side Component Diagram. 50
4.4 UC-IC Collaboration Class Diagram. 51
4.5 Two UC-IC Domains Communicating Through SIP. 52
4.6 UC-IC Client Platform Components. 54
4.7 UC-IC Client Side Class Diagram. 55
4.8 JIP Application Design. 57
4.9 JIP Geocoding Sequence. 59
4.10 Watch Together Server Class Diagram. 62
4.11 Watch Together Connection Message Sequence Chart. 63
4.12 Watch Together Invitation Message Sequence Chart. 64
4.13 Watch Together Client Activity Diagram. 68
4.14 Watch Together Client Interfaces. 69
4.15 Google Maps Application Design. 72
4.16 Subscriber Class Diagram. 74

4.17 Subscriber Message Sequence Chart. 74
4.18 Publisher/Subscriber Server Message Sequence Chart. 77
4.19 Smart Publisher Class Diagram. 78
4.20 Smart Publisher Message Sequence Chart. 80
4.21 Sensor Feeder Class Diagram. 82
4.22 Sensor Feeder Message Sequence Chart. 83
4.23 Sensor Registry Class Diagram. 85
4.24 Sensor Registry Message Sequence Chart. 85

5.1 UC-IC Collaborative Environment. 87
5.2 UC-IC Photo Viewer Collaborative Session. 88
5.3 UC-IC Environment with JIP Windows Open. 89
5.4 Two Users Collaborating on a JIP Map Window. 89
5.5 Graph of Sensor Data Values from Table 5.1. 91
5.6 Rules-Based Colouring of Sensor Data from Table 5.1. 91
5.7 Six Facebook Users Collaboratively Watching a YouTube Video. . . 93
5.8 Map Sharing and Video Chat. 94
5.9 Watch Together Map Application Sensors List. 95
5.10 NOAA Sensors in Google Maps Application. 96
5.11 Real-Time GPS-Based Vehicle Movement in JIP. 97
5.12 Real-Time Vehicle Tracking From Desktop and iPhone. 97
5.13 Real-Time Latitude/Longitude Data in JIP Info Window. 98
5.14 Real-Time Flight Simulator Data Inside JIP. 99
5.15 Real-Time Marker Update from Engine Failure. 99

List of Tables

5.1 Sensor Data Values from Four Locations. 90

Abbreviations

ACM	**A**ssociation for **C**omputing **M**achinery
AJAX	**A**synchronous **J**avascript **A**nd **X**ML
API	**A**pplication **P**rogramming **I**nterface
CSS	**C**ascading **S**tyle **S**heets
DHTML	**D**ynamic **H**yper **T**ext **M**arkup **L**anguage
EJB	**E**nterprise **J**ava**B**eans
ESRI	**E**nvironmental **S**ystems **R**esearch **I**nstitute
GIS	**G**eographic **I**nformation **S**ystem
GML	**G**eographic **M**arkup **L**anguage
GNSS	**G**lobal **N**avigation **S**atellite **S**ystem
GPRS	**G**eneral **P**acket **R**adio **S**ervice
GPS	**G**lobal **P**ositioning **S**ystem
HTML	**H**yper **T**ext **M**arkup **L**anguage
IEEE	**I**nstitute of **E**lectrical and **E**lectronics **E**ngineers
HTTP	**H**yper**T**ext **T**ransfer **P**rotocol
IETF	**I**nternet **E**ngineering **T**ask **F**orce
JDBC	**J**ava **D**ata**B**ase **C**onnectivity
JNDI	**J**ava **N**aming and **D**irectory **I**nterface
JEE	**J**ava Platform, **E**nterprise **E**dition
JIP	**J**oint **I**ntelligence **P**icture
JMS	**J**ava **M**essage **S**ervice
JSF	**J**ava **S**erver **F**aces
MOM	**M**essage-**O**riented **M**iddleware
MUC	**M**ulti-**U**ser **C**hat

NASA	**N**ational **A**eronautics and **S**pace **A**dministration
NCCT	**N**etwork **C**omputing and **C**ontrol **T**echnologies
NTRIP	**N**etworked **T**ransport of **R**TCM via **I**nternet **P**rotocol
O&M	**O**bservations & **M**easurements
OGC	**O**pen **G**eospatial **C**onsortium
ORB	**O**bject **R**equest **B**roker
PDA	**P**ersonal **D**igital **A**ssistant
REST	**RE**presentative **S**tate **T**ransfer
RIA	**R**ich **I**nternet **A**pplication
RDBMS	**R**elational **D**ata**B**ase **M**anagement **S**ystem
RPC	**R**emote **P**rocedure **C**all
RTC	**R**eal **T**ime **C**ollaboration
RTCM	**R**adio **T**echnical **C**ommission for **M**aritime Services
RTMP	**R**eal **T**ime **M**essaging **P**rotocol
SAS	**S**ensor **A**lert **S**ervice
SES	**S**ensor **E**vent **S**ervice
SIP	**S**ession **I**nitiation **P**rotocol
SOAP	**S**imple **O**bject **A**ccess **P**rotocol
SOA	**S**ervice-**O**riented **A**rchitecture
SOS	**S**ensor **O**bservation **S**ervice
SPS	**S**ensor **P**lanning **S**ervice
TCP/IP	**T**ransmission **C**ontrol **P**rotocol/**I**nternet **P**rotocol
UDP	**U**ser **D**atagram **P**rotocol
W3C	**W**orld **W**ide **W**eb **C**onsortium
WFS	**W**eb **F**eature **S**ervice
WMS	**W**eb **M**ap **S**ervice
XML	e**X**tensible **M**arkup **L**anguage
XMPP	e**X**tensible **M**essaging and **P**resence **P**rotocol

Dedicated to my parents and my brother

Chapter 1

Introduction

1.1 Motivation and Research Objectives

Web 2.0 has dramatically transformed the way in which information is collected and presented to online users, and the enormous popularity of social networking has created a growing appetite for real-time data. Users are no longer satisfied with just viewing simple HTTP pages but expect the ability to share and collaborate with other people, such as friends or colleagues, online and in real-time. The social networking website Twitter [1] has seen a 200% increase in users from 2009 to 2010 [2] by providing users with a simple way to share real-time data. The geosocial website Foursquare is currently growing at a rate of 25,000 users per day [3]. Location information and other Geographic Information Systems (GIS) data is an increasingly important part of what users wish to share and view online, yet real-time sensor data distribution on the web remains relatively primitive, especially when considered in a collaborative context.

Google Maps is an example of how Web 2.0 technologies such as AJAX can be used to create online map services that are easy to access, user-friendly, and fast. Thanks to flexible web-based mapping APIs, it is now possible for non-experts to plot and distribute GIS (Geographic Information System) data to a large audience. For example, users can view “mashups” of geographical data such as housing information from the website Craigslist plotted on a Google Map [4]. New web browser technologies have made it easy for organizations and users to access and publish dynamic data, such as what they are doing and where they are currently located.

Shipping services like FedEx and UPS now provide users with information on the location of their package as it reaches various locations. In the United States, Domino's Pizza offers a Pizza Tracker on their website, where customers can track the status of their pizza order online as it goes from "order placed" to "prep" to "bake" to "quality check" and finally to "out for delivery". The Domino's service even tells users the estimated wait times, the name of the pizza maker and the name of the delivery person. "I think I get more enjoyment out of watching pizza tracker than eating the pizza", stated one user [5]. Companies like FedEx, UPS and Domino's Pizza (along with many others) have recognized the importance of providing customers with up-to-date information by using the internet. With a standard real-time mapping architecture, these companies can go even further and provide users with the locations of the delivery vehicles. Although there may be privacy challenges to overcome, these services would be very valuable to customers, and the technology now exists to make real-time web-based mapping possible.

While there exist many online services that promise "real-time" geographic data, a closer look will almost always reveal that the data is significantly delayed. For example, Google Maps provides a button in the top-right corner of their interface that claims to provide "real-time traffic data" by showing green, yellow and red overlays to indicate the traffic conditions of major roads. While this information is very useful, users have found that it is often delayed by up to 15 minutes [6]. Even Google Latitude [7], a service designed to let users track friends who have opted to share their GPS location from their mobile devices, is generally delayed by approximately 5 minutes [8].

Another important aspect of making real-time data accessible to users via the web is how the data is presented to the user. The ships on the MarineTraffic website [9] do not move on the map in an animated fashion; instead, the user is required to click a "refresh now" button so that each ship can be drawn at its updated location. Web technology such as Adobe Flash [10] now allows for detailed animations and effects to be rendered within the user's web browser. This makes it possible to have smooth movement animations of map markers, as well as to display detailed dials and graphs when the markers are clicked on. In addition, streaming video and audio from remote camera feeds can be inserted into maps. So far, the typical web page layout has limited the interaction possibilities for online maps when compared with windowed desktop applications. JavaScript-based web technologies now allow web maps to be separated into desktop-like windows, meaning that multiple map

Figure 1.1: LINET Sensor Network.

instances can be opened within a web page and maps can be arranged such that all real-time information the user needs is clearly visible. These and other types of visualizations are important for creating a graphical user interface that clearly communicates various GIS data to the user.

Data collected from remote sensor networks is important in the decision-making process of scientists and governments for uses such as preserving the environment. Temperature, humidity, pressure, toxicity and other types of sensor data can reveal significant trends in climate change and the living conditions of an urban area. Recently, the monitoring of carbon dioxide emissions has been of particular global interest [11]. A variety of organizations, such as Germany-based Nowcast [12], have therefore deployed atmospheric sensors. Nowcast's LINET is an example of a sensor network for monitoring atmospheric data with over a hundred sensors for lightning tracking distributed across Europe, as shown in Figure 1.1. It is therefore increasingly important for GIS applications to ensure that the attention of all people involved is on the most relevant and recent sensor data. This implies a collaborative system that is able to access a variety of sensor networks and display real-time data in a way appropriate for monitoring and analysis.

Real-Time Collaboration (RTC) has shown its benefits to productivity, communication and cost-savings. Commercial products [13] have opened doors to new paradigms of teamwork which have considerably gained in popularity over the past few years. Increasingly, web-based technologies are used for collaborating

with other people. Websites like Facebook offer the ability for online content sharing between users, though a message posted by a user containing links to photos or videos is later viewed by one or more users separately. Google Docs [14] allows multiple users to work on the same document at the same time, and the document is stored on Google's remote servers in the cloud. Users are not able to collaborate over maps, however. In addition, collaborating with others in real-time on maps themselves containing real-time sensor data has yet to be attempted from inside a web-browser.

This thesis proposes a standard architecture that uses a collaborative web-based environment to monitor and display data from multiple real-time sensor networks. In this thesis, "standard architecture" is used to refer to an architecture that is open, reusable, and based on established standards. The architecture builds on the publisher/subscriber messaging model to meet the data accessibility and scalability requirements of collaborative online environments while offering support for real-time sensor data that can be presented to the user in useful ways. The architecture can be adapted to many other client and server platforms, but the research presented here will focus on a web-based client side (using technologies such as Adobe Flash and AJAX) driven by application servers based on the Java Platform Enterprise Edition (JEE). The sensor data used already conforms to the Sensor Observation Service (SOS) standard or is otherwise converted to this flexible and open sensor data format. SOS is one of the standards of the Open Geospatial Consortium (OGC), an international standards organization with over 400 supporting companies and institutions [15].

Acquired sensor network data will be rendered from the web-browser by using two in-house platforms for web-based collaboration. Both platforms were developed as part of the NCCT research group, with the contributions of the author of this thesis focusing on the sensor and GIS features of the platforms. The first is an AJAX-based real-time platform for multi-domain collaboration known as UC-IC ("you see I see"), which recreates a familiar desktop environment within the web-browser, allowing maps containing sensor data to be organized within windows and manipulated in user-friendly ways. The second is a Flash-based real-time platform for collaborative multimedia known as Watch Together. This platform makes it possible to do more complex animations and use webcam chat while sensor data is displayed among multiple synchronized users. Both platforms were designed from the ground-up to provide a social environment that enables real-time transfer of

applications and information to and from collaborators. This thesis will show how these platforms allow for novel ways of viewing and interacting with real-time data from sensor networks among multiple concurrent users. Live data from Microsoft Flight Simulator 2004 [16] is used to evaluate their real-time performance.

Unlike remote desktop solutions, the collaboration discussed in this thesis involves the sending and receiving of an application's complete program logic as well as its data. This is a novel approach to web-based collaboration that has not been attempted before. For example, by sending a map application, the receiving users will not only receive the map application logic, but will also see the map centered at the same location where it was for the sender. If the map contained a video, users are able to collaborate on the video data while having the video rendered on each user's local machine. Videos are not re-encoded as part of a remote screen update; they therefore run at full speed for all users and synchronization is ensured through event-based signals.

So far, researchers have tackled the problem of making real-time data available within the web browser in a variety of ways. Unfortunately, most existing academic solutions do not target mainstream users outside of the lab and therefore rely on custom browser plugins and non-standard components. The architecture proposed in this thesis aims to make real-time GIS data as accessible as possible for users. As such, it must make use of the latest web-based technologies and standards. In order to achieve this, the system must not require the download and installation of proprietary browser plugins and must run on various types of devices, including emerging smartphones and tablet PCs. For UC-IC, this means using standard JavaScript and AJAX code which can be displayed by all modern browsers, along with a JEE-based server receiving real-time data through the Java Message Service (JMS) standard. For Watch Together, this is achieved through the use of the Flex 4 framework for Adobe Flash [17] on the client side and a Real Time Messaging Protocol (RTMP) based [18] server. While Adobe Flash is a browser plugin, it can be found on more than 98.9% of client machines [19]. Currently, this represents better penetration than HTML5, which is still in the course of development and can be found only in experimental states in all major web browsers.

1.2 Thesis Contributions

This thesis contains a number of contributions to the fields of real-time web GIS and collaborative systems as presented in [20], [21], [22], [23], [24], [25], [26]:

1. A publisher/subscriber-based architecture for real-time data is presented which defines the protocols and standard practices required for streaming real-time GIS data to a web browser. By standardizing the architecture, this thesis provides researchers and organizations with guidelines to ensure that they are setting up a reliable push-based GIS data solution for online data distribution. It also provides the necessary real-time delivery mechanisms and adaptability required for the effective delivery of real-time GIS data to a web browser. Finally, the architecture is designed to support integration with the latest social networking and online collaboration platforms for synchronized multi-user collaboration over GIS data.

2. A real-time mechanism is given that uses the Sensor Observation Service (SOS) GIS standard and open source projects to allow access to any real-time sensor data source through an XML-based format ideal for real-time communication. This includes data ranging from simple weather stations to entire sensor networks. By adapting GIS data to the SOS format, this thesis builds on established components and proven open standards for GIS data which have well-defined methods for storing values such as temperatures and locations. This also allows the system to have access to the large amount of real-time data already in this format from trusted sources such as NOAA [27] and other existing online sensor webs.

3. A real-time platform for multi-domain collaboration is implemented that allows multiple instances of map applications showing various real-time data to be open at the same time and instantly shared with different users to ensure they have the information they need. Known as UC-IC, the platform allows users to send, share and distribute control over applications and their data in real-time. UC-IC was developed as a project with others at the NCCT research lab and uses JavaScript and AJAX to provide a desktop-like environment within a typical web browser. Support for collaboration is built directly into the user's workspace by using an extension to the Session

Initiation Protocol (SIP) to establish sessions. It thus enables a user to participate in several collaboration sessions concurrently and to use applications in more than one session at a time.

4. A real-time platform for collaborative multimedia is implemented that uses Adobe Flash to allow the use of webcam chat while collaborating over maps, videos, photos, documents, listening to music, and playing games, all while staying synchronized with the other users. Known as Watch Together, the project was developed along with others at the NCCT research lab to allow the sharing of web-based content with colleagues and friends in real-time. The map application takes advantage of Watch Together's Flash-based nature by providing smooth animations for rich real-time content. Watch Together can also easily be extended by developers to support new sources of online media, such as different map providers.

5. A general simulation mechanism for demonstrating the propagation of real-time data throughout a real-time architecture is given. The mechanism works by using a flight simulator game to provide real-time sensor data for display on the web. Developed as a project with others at the NCCT research lab, a special driver was connected to the flight simulator that can access the live airplane data (such as altitude and heading) as a user is controlling the plane. This data can then be displayed by a real-time collaborative web client for observation of delays between the in-game actions and the corresponding updates in the web browser.

1.3 Thesis Organization

This thesis proposes a web-based solution that allows real-time access to sensor data for multi-user web-based collaboration via a publisher/subscriber architecture. The related work in the fields of web GIS and collaborative systems are presented first and the architecture requirements are identified. A high-level overview of the architecture is then given, along with detailed specifications that are required for the delivery of real-time GIS data to collaborative browser-based environments. A functional implementation of the architecture is then presented, and it shown to function with two different collaborative web applications and

multiple different sensor sources. The results are then presented and necessary conclusions are drawn. As such, the rest of the thesis is organized as follows:

Chapter 2 presents the need for an architecture for real-time web-based collaborative mapping by providing the necessary background and identifying various related work and important literature for both real-time web GIS and real-time web collaboration problems.

Chapter 3 describes the high level design and requirements of the proposed architecture for optimal real-time data delivery to collaborative web-based environments. It maps the requirements for the system to the architecture's main components.

Chapter 4 contains the low level architecture and implementation details that make it possible to deliver real-time GIS data to a web browser. Various mechanisms are described for the necessary accessibility and performance characteristics to be met.

Chapter 5 provides the results obtained from a number of tests, including real-time data from a flight simulator game being displayed within two collaborative web-based environments, namely UC-IC and Watch Together.

Finally, Chapter 6 presents conclusions on the benefits provided by the proposed solution. Future developments of the architecture and implemented systems are also described.

Chapter 2

Background

This chapter presents the current state of real-time Geographic Information System (GIS) data on the Internet. It then explains the need for a new architecture for real-time data delivery to a collaborative web-based client by examining existing academic and commercial solutions.

2.1 Web Mapping Technology Overview

Ever since its release to the public in 1993, the Internet has enjoyed large amounts of growth and innovation [28]. With the level of interactivity now possible on the web, users are no longer browsing simple "web pages", but rather sophisticated "web-based applications". These web applications may perform complex server-side services that give users quick access to the features they expect of a full desktop application, all without the users having to leave their web browser.

This fundamental shift in the workings of the Internet over the last few years has brought about the term "Web 2.0", which was first coined by publisher Tim O'Reilly at the O'Reilly Media Web 2.0 conference in 2004 [29]. Besides the concept of "social networking" that encourages online collaboration between users, Web 2.0 has also grown to include developer-friendly application programming interfaces (APIs) to allow for powerful combinations of data from multiple online sources and reuse of existing components to form "mashups", thereby ensuring continuous progress and enhancement of what the web has to offer.

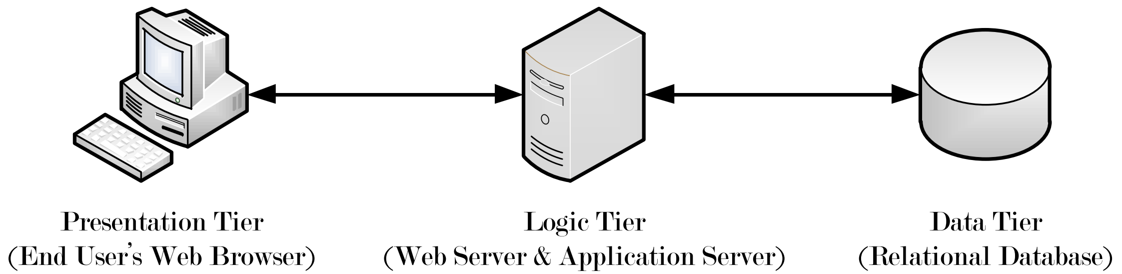

Figure 2.1: Typical Three-Tier Architecture.

To make this new level of interactivity possible, a multi-tier software architecture was adopted that separated the client-server communication into three separate tiers: the presentation tier, the logic tier and the data tier [30]. This three-tier software architecture can be seen in Figure 2.1. The three-tier model requires software to be modular and have well-defined interfaces. This modularity gives a great deal of flexibility when it comes to choosing the right web application technology since each tier can be upgraded or changed independently of the others.

The end user gains access to the web application through a web browser. The web browser displays data and lets the user interact with the web application via a web page. Although there are several exceptions, the web browser generally displays content that conforms to standards defined by the World Wide Web Consortium (W3C) [31]. This includes the Hyper Text Markup Language (HTML) standard for defining the web page structure, as well as Dydamic HTML (DHTML) standards such as JavaScript and Cascading Style Sheets (CSS).

JavaScript is a client-side scripting language that works with the HTML contents of a page to make them more dynamic, such as by allowing button graphics to change when the user moves their mouse cursor over them. CSS is a style sheet language that gives control over page formatting and presentation. All of these elements make up the presentation tier of the three-tier architecture.

Based on user actions, the web browser requests content by sending a Hypertext Transfer Protocol (HTTP) request to the logic tier. The logic tier consists of a web server and an application server. The web server is responsible for serving the web browser with HTTP responses along with HTML documents and other data such as image files. The web server works alongside a web application server that handles the main processing and generation of HTML content.

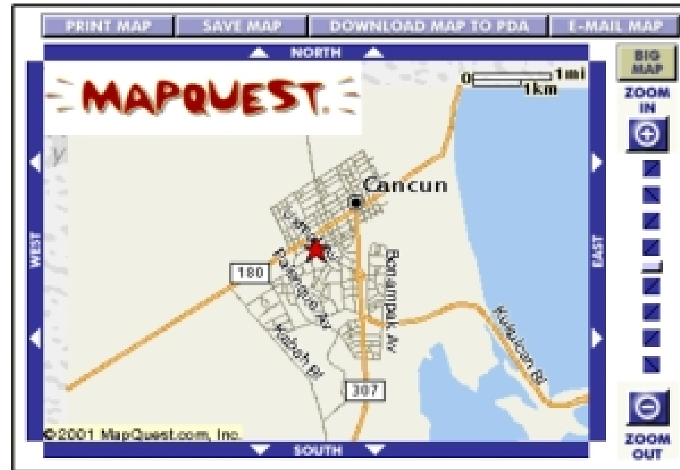

FIGURE 2.2: Classic MapQuest Interface.

The data tier is responsible for retrieving data for the logic tier. A relational database management system (RDBMS) is typically used for its fast, indexed access to data, as well as its ability to maintain the database in a consistent state during service outages. As a separate tier, the data tier keeps data neutral from the access methods in the logic tier and offers scalability and flexibility for meeting ever-changing data storage requirements.

To illustrate how the three-tier architecture works, the map website MapQuest will be taken as an example. Released in 1996 as a free service, MapQuest [32] made it possible for users to search for a location by name and to navigate the resulting map by using several buttons that surrounded the static map image (including buttons to select the zoom level). This early MapQuest web interface is shown in Figure 2.2.

To access this version of MapQuest, a user will use a web browser to contact the MapQuest web server and retrieve a web page containing a text box and a *Search* button (among other things). By entering a city name and clicking the button, the user sends a HTTP request containing the city name to the logic tier of the MapQuest service. The application server of MapQuest's logic tier then uses the geographical data stored in the data tier to generate an image file of the specific address requested, which is sent back to the web browser by the web server as part of a new web page. The data tier for such a web application could contain extensive geographical information for the entire world, but the user is only requesting a very small subset of it at a time, and always does so through the logic tier.

For MapQuest, the fact that HTTP responses contained full web pages meant that users would see the entire web page refresh for each search or navigation operation. Nevertheless, even with this inconvenience, MapQuest would show how the interactivity and accessibility brought forth by the Internet made it a good fit for geographic information.

The Internet evolved to support more dynamic web page content through the introduction of Rich Internet Applications (RIA) and Web 2.0 technologies for the presentation layer. Perhaps the most recognized web technology associated with Web 2.0 is AJAX, which stands for Asynchronous JavaScript and XML. Over the years, several improvements have been made to JavaScript through the addition of new libraries. AJAX is one of these libraries, adding an important new object called XMLHttpRequest. This object allows web pages to exchange XML (eXtensible Markup Language) data even after a web page has finished loading. The received XML data can then trigger JavaScript code to modify the contents of the web page without having to refresh the page itself. The load on the logic tier is therefore reduced since only the information that needs to be updated is transferred rather than the entire web page [33].

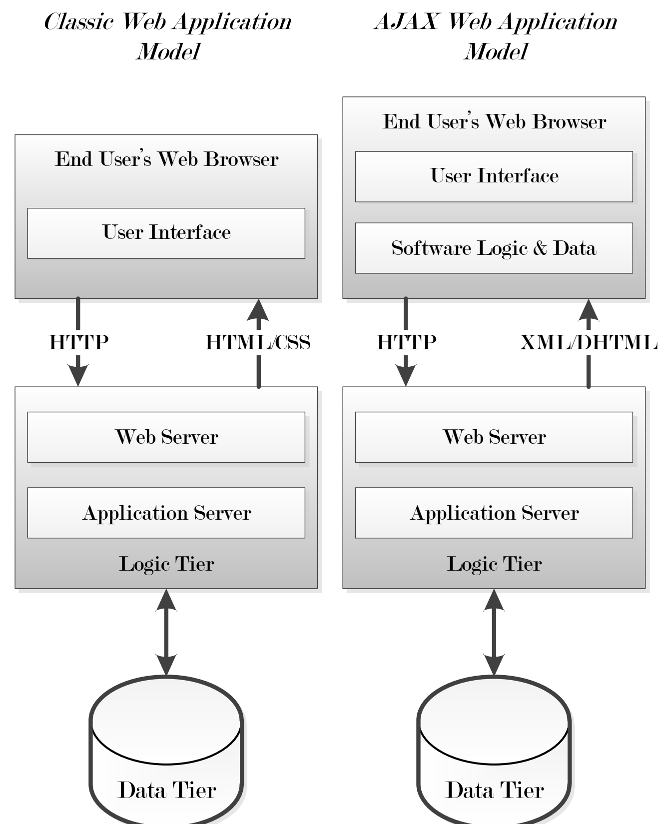

FIGURE 2.3: Classic vs. AJAX Application Model.

Figure 2.3 shows the difference between the classic web application model and the more recent AJAX web application model. The classic web application model keeps the software logic and data on the server-side and makes them accessible to the user via HTML/CSS responses to the browser's HTTP requests. The AJAX web application model increasingly places the software logic and data on the client side and allows the web server to transfer new software logic and data (such as DHTML and JavaScript packaged within XML data) even after page loading has completed. This makes it possible to create much more interactive and data-driven experiences that users can instantly gain access to with just a web browser and an internet connection.

One of the first websites to make the most out of AJAX technology was a mapping site named Google Maps. Launched in 2004, Google Maps combined visually appealing maps with a very accessible user interface [34]. It used AJAX to dynamically load sections of the map as the user dragged the map with the mouse cursor, a defining characteristic of what are now known as "slippy maps" [35]. This user-friendly map presentation technique would later be adopted by most map websites, including MapQuest in 2007 [36].

As user demand for web-based applications continued to increase, limitations with the HTML/JavaScript/AJAX technologies became apparent and required different methods of displaying content within a browser in the form of plugins. The Adobe Flash [10] browser plugin allows web browsers to display detailed animations and streaming multimedia content within a web page. Although Flash is not a true web standard, Adobe claims that Flash is present on 98.9% of client machines [19]. Development of interactive Flash components is done using a language similar to JavaScript known as ActionScript. Once compiled, the final Flash file is uploaded to a web server and embedded inside a web page via HTML code. Flash components within a web page can also send and receive data from a server using a system similar to AJAX, whereby an embedded Flash object is able to use Adobe's proprietary Real Time Messaging Protocol (RTMP) [18] to communicate with a "media server". This allows the data within Flash applications to be updated dynamically and combined with other multimedia elements such as webcam chat. The Adobe Flex SDK is particularly useful for such Flash-based RIA development [17]. In 2008, Google released a Flash version of their popular Google Maps API to allow developers to take advantage of these capabilities [37].

2.2 The Need for an Architecture

Since the release of Google Maps and its flexible web mapping API, web-based mapping has received a significant amount of attention from the research community. Most of this research has been focused on providing web users with visualizations for geographic data that was previously more difficult to represent effectively online. By using the three-tier architecture, large databases of GIS data that were previously only accessible through sophisticated desktop-based software could now be made available to a mainstream audience through the web. The AJAX-driven or Flash-driven presentation tier dynamically loads map sections as the user drags the map, allowing for quick and intuitive access to the information the user requires. The logic tier contains powerful application servers for performing geographic computations (such as the optimal path between two locations) on large data sets and for dynamically generating the images required by the presentation tier. The data tier for web mapping applications is powered by spatial versions of relational databases that are optimized for the unique indexing requirements of geographic data. These advanced analytical capabilities and accessible delivery mechanisms have allowed web mapping to advance from static web cartography images to elaborate and multifaceted web GIS applications.

A variety of web mapping standards and technologies emerged to make such web mapping possible. While a variety of proprietary and commercial technologies exist from companies like ESRI [38], a major problem in web-based mapping has been the lack of interoperability between different component vendors and data standards [39]. Using a spatial database from one provider and the application server from another provider may cause problems if the two systems are incapable of interoperating. For example, the application server may not integrate properly with the spatial database, meaning that it cannot fulfill the geographical operations requested by the client. The GIS data may originate from a sensor in a format that is not recognized by a particular spatial database or application server. Finally, the client application may not be able to display the geographic data in the format provided by the application server. While several approaches exist to reduce this problem, they generally involve computationally-expensive encoding or conversion operations at multiple stages; something which is hardly ideal for real-time data. A standard is therefore needed that is flexible enough to represent many types of GIS data and that can be implemented in a manner appropriate for real-time delivery to a web browser (non-verbose, quick to process, etc.).

While dynamically loaded map section techniques such as those used by Google Maps are a step in the right direction, web mapping is generally not making use of the latest web technology to its full potential. Map websites will promise "live traffic" and "real-time tracking", when, in fact, in almost all cases the data is delayed by several minutes and does not update inside the web browser in real-time. The website MarineTraffic [9], a project of the University of the Aegean, Greece, shows a "live" map containing the positions of different ships around the world. However, the page contains a timer that refreshes the data every minute, meaning that ships do not animate as they move across the map in real-time. In addition, a look at the fine print at the bottom of the page reveals that "Vessel positions may be up to one hour old or incomplete". One must therefore take any claims of "live" or "real-time" maps with a high degree of skepticism, as the claims are almost always false. From an architecture point of view, this indicates a shortcoming with the real-time support of the data tier that the logic tier is accessing, as well as with the methodology used by the logic tier to update the presentation tier.

A related limitation of current approaches to online maps is the lack of social features for real-time collaboration. Over 500 million people now have a presence on the Internet through the social networking website Facebook. In fact, over a third of Canada's population now has a Facebook account [40]. In an article on the influential Web 2.0 blog TechCrunch [41], David Sacks argues that information should be "pushed" by a user's network of friends or co-workers instead of the user having to search for a term and filter the results. Since friends and co-workers will push information that is likely to be useful, users can "access a world of information that is both increasingly comprehensive and personal to them". The substantial growth in social networking and demand for collaborative features cannot be ignored when it comes to web-based mapping, especially since social networking already revolves around GIS data such as where someone is from and where an organized event should take place. Currently, users generally do not have access to a list of their friends or co-workers when they visit a mapping website. In addition, existing research has often failed to recognize the importance of this emerging trend and has not connected it properly with web mapping.

Another problem with existing online mapping solutions is that their interfaces limit the interaction possibilities that users have when using maps. Offline GIS applications like ArcGIS from the Environmental Systems Research Institute (ESRI)

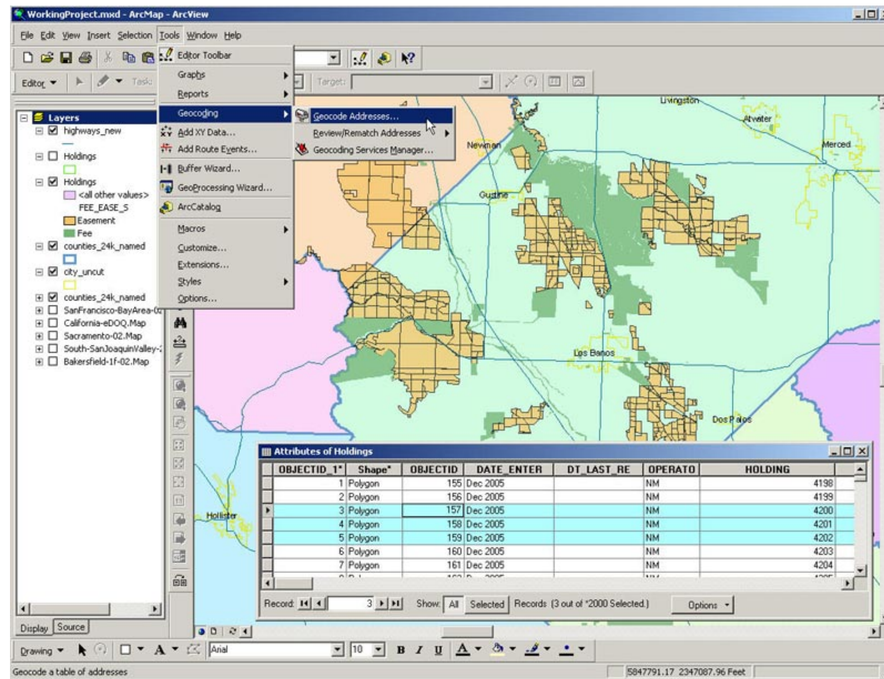

FIGURE 2.4: Multiple Windows and Menus in ESRI's ArcGIS.

[38] offer flexible dragging and dropping of data between multiple inner windows containing maps. Windows can be positioned on top of other windows and many features are accessible from menus, as shown in Figure 2.4. Most online mapping solutions partition the web page into immobile sections and do not allow more than one map instance to be created. Attempts to add more advanced functionality often require obscure browser plugins that need to be downloaded and installed for the user's specific operating system and browser. Existing literature has also generally focused on the mechanisms for delivering GIS data, without placing much importance on ensuring that the user interface is accessible to a nonspecialist audience and that it offers a good overall end-user experience.

The problems above can be addressed with a flexible architecture that uses lightweight open GIS standards for delivering real-time geographical data to a collaborative web-based environment. The architecture must be able to adapt the data from a large variety of sensor sources and networks such that it can be optimally displayed within a regular web-browser. Such an architecture would also allow users to share and collaborate on the data with their friends and co-workers by using a flexible and user-friendly interface that is focused on addressing the latest needs of web users. In addition, what is needed is an open architecture that is based on the three-tier design model and uses the latest web-based standards to allow real-time GIS data to be delivered to end users on practically any web-enabled

device.

2.3 Related Work

Web mapping and collaborative computing have been important areas of research in recent years, with much attention given from publications and conferences supported by IEEE, ACM and Springer. The research has included various approaches for solving the problems identified. Solutions for supporting a variety of GIS data include standards from bodies such as RTCM and OGC. Solutions for real-time data delivery to a web browser include publisher/subscriber mechanisms such as XMPP and JMS. Finally, several collaborative technologies and web-based mapping user interface approaches exist in the current literature. While the proposed architecture must meet the needs of the various existing solutions, it must also be able to integrate with a large assortment of existing components by being based on open standards. This section is composed of related work on real-time web-based mapping architectures and collaborative systems, including related products, related standards, and related technologies.

2.3.1 Web Mapping Standards

With the growing demand for web-based GIS data, a variety of standards have emerged to try to resolve the interoperability problem between components of different vendors while still providing the light-weight formats required of web clients. This section looks at some of the methods from existing research of representing GIS data for real-time web-based use.

2.3.1.1 RTCM Standards

The Radio Technical Commission for Maritime Services (RTCM) format for real-time collection and exchange of GIS data via the Internet was explored by Weber, Dettmering and Gebhard in [42]. RTCM was established by the international standards organization of the same name to encode a variety of precise position and radar-related data, including Global Navigation Satellite System (GNSS) data

such as via the satellite Global Positioning System (GPS), in a format appropriate for streaming. More specifically, the standard is known as RTCM Special Committee 104 (RTCM-104). Weber et. al. showed how GNSS data can be streamed via HTTP to personal digital assistants (PDAs) on mobile IP-Networks such as General Packet Radio Service (GPRS). They called their technique Networked Transport of RTCM via Internet Protocol (NTRIP). NTRIP Servers would receive real-time data from certain devices and broadcast this data to NTRIP Clients via a NTRIP Caster. They found that the TCP/IP (Transmission Control Protocol/Internet Protocol) protocol suite is highly preferred for reliable real-time transfer of geographical data between the server and mobile IP clients since the User Datagram Protocol (UDP) is prone to packet loss. They also found that the RTCM-104 data format was light enough for transfer via the mobile IP network and generally observed "latencies of the order of less than three to four seconds" in their implementation.

The RTCM-104 format itself consists of messages containing a sequence of up to 33 words, each word having a length of 30 bits. The last six bits of each word are used for parity (something of little value when using TCP/IP-based communication). The first two words contain header information (message type, station id etc. in a well-defined order), while the rest may contain the actual data. Weber et. al. therefore developed a native NTRIP Client application that processed the RTCM data directly on the PDA, rather than in the PDA's web browser.

Heo, Lim and Rizos explored the feasibility of browser-based real-time data delivery of GNSS data conforming to the RTCM format [43]. They used the NTRIP technique to transmit navigation data from various NTRIP casters set up at various reference stations. Their JEE-based application server would receive this data and store it a RDBMS database, as well as transmit it to connected clients. The client application was a Java Applet running in the user's browser, which was found to receive data once per second. Historical data could be accessed from the RDBMS for additional client-side analysis.

While the RTCM standard supports a variety of positioning-related data, its binary nature is not ideal for web-based use, requiring a less-accessible Java Applet-based solution for streaming and processing in real-time. Heo et. al. praised their Java-based implementation for its platform independence, yet the adoption of the Java browser plugin on the client side web browser is still far below standard web technologies such as AJAX and competing browser plugins such as Adobe Flash

[19]. XML-based data is more ideal for processing with these more accepted web technologies. The RTCM standard is also not intended to handle GIS information beyond position data, making its use less than ideal when a large variety of sensor data is to be supported. Nevertheless, the experiments of Weber et. al. and Heo et. al. have successfully demonstrated many of the key concepts involved in delivering real-time sensor data to mobile and browser-based clients over an IP-based network.

2.3.1.2 OGC Standards

Another approach for managing various GIS data was recently investigated by Mayer et al. from the University of Bonn [44], who researched the architectural implications of retrieving and displaying dynamic data in a standardized way. They found that the Sensor Web Enablement (SWE) set of standards of the Open Geospatial Consortium (OGC) provided the ideal data exchange standards and interfaces for bringing GIS data to the web. The goal of the SWE standards is to allow all possible types of sensors to be accessible, detectable and controllable via the web in an open and standardized way [45]. Specifically, they relied on the Sensor Observation Service (SOS) standard and Observations & Measurements (O&M) standard to connect to a real-time traffic data source and make this data available within a web browser.

The SOS standard allows aggregating live sensor data and making it available as a web-based service. Specifically, it defines the REST-based interface that an SOS server must provide [46]. This includes the XML-based syntax required to add a new sensor with available data (*RegisterSensor*), to submit new real-time data for the sensor (*InsertObservation*), to retrieve detailed information on available sensors (*GetCapabilities*), and to retrieve the latest data from a specific sensor (*GetObservation*), among others. Since SOS servers provide REST-based (sometimes called *RESTful*) services, HTTP GET and HTTP POST is used to submit and retrieve the necessary data (based on the specifications of the standard) via a URL. The following URL is an example of sending a *GetCapabilities* request to the SOS server hosted by the National Oceanic and Atmospheric Administration (NOAA) in the United States, which returns an XML file containing information about all live sensor data they provide:

```
http://opendap.co-ops.nos.noaa.gov/ioos-dif-sos-test/SOS?service=SOS&request=GetCapabilities
```

As Mayer et al. discovered, the strength of the SOS standard lies in its dependence on other well established open interfaces and protocols from OGC. For example, OGC has a flexible way of describing sensor data and formats: the Observations & Measurements (O&M) standard. O&M defines an “observation” as an event or action which produces a *result* whose value is an estimate of a *observedProperty* of a *featureOfInterest* obtained via a *procedure* [47]. These relationships are represented via UML in Figure 2.5. This model serves as the foundation of how observations are formed in the *InsertObservation* requests to SOS servers and *GetObservation* responses received from SOS servers.

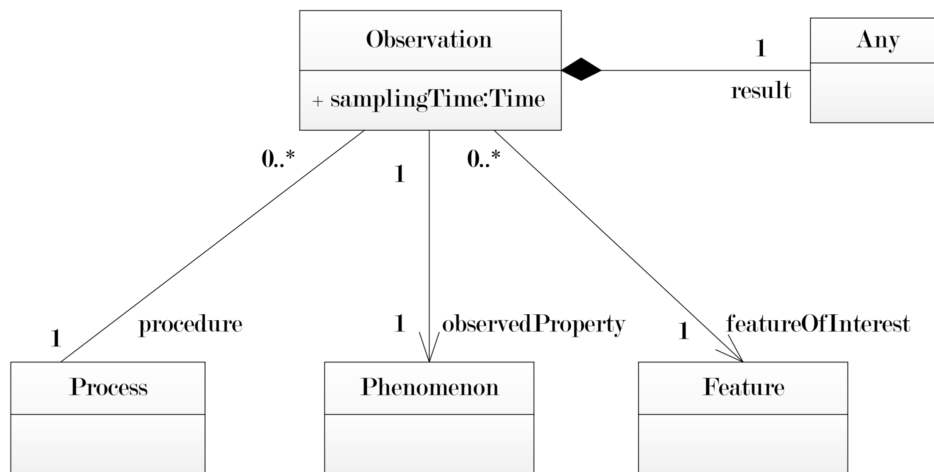

FIGURE 2.5: Observations & Measurements Model.

Although the format was designed for GIS data, the flexibility of the O&M format will now be illustrated by using a non-GIS example. To add the observation “Specimen H39 was determined on 2010-10-14 by Cristian Gadea to be of the species Eucalyptus Caesi”, “Specimen H39” is identified as the *featureOfInterest* (with “Specimen” as the *Feature*), “Cristian Gadea” as the *procedure*, a “taxon” as the *observedProperty*, and “Eucalyptus Caesi” as the *result*. In addition, we know the *samplingTime* of the Observation to be “2010-10-14”. A rough outline of the XML required to add the observed values to the SOS server is shown in Listing 2.1, with only the *samplingTime* value shown in full. In this case, the Geographic Markup Language (GML) standard, another OGC standard, is used to define the time value [48].

```xml
<?xml version="1.0" encoding="UTF-8"?>
<InsertObservation service="SOS" version="1.0.0" xmlns=...>
<AssignedSensorId>urn:ogc:object:Sensor:MyOrg:12349</AssignedSensorId>
<om:Observation>
   <om:samplingTime>
      <gml:TimeInstant>
         <gml:timePosition>2010-10-14T00:00:00Z</gml:timePosition>
      </gml:TimeInstant>
   </om:samplingTime>
   <om:procedure ... />
   <om:observedProperty ... />
   <om:featureOfInterest ... />
   <om:result ... />
</om:Observation>
</InsertObservation>
```

Listing 2.1: Outline of Sample InsertObservation Request.

Other noteworthy standards that make up the OGC SWE initiative are the Sensor Model Language (SensorML, an XML encoding for describing the sensors and measurement *procedures*), Transducer Markup Language (TML, an XML schema for describing transducers), Sensor Planning Service (SPS, for user-driven acquisitions), and Sensor Alert Service (SAS, a standard for publishing and subscribing to alerts based on sensor data). Other noteworthy OGC standards include Web Map Service (WMS, a protocol for retrieving map images generated from geographic data) and Web Feature Service (WFS, a protocol for performing queries on geographic data).

In the implementation from the University of Bonn, the SOS server was connected to live traffic data provided over a public radio signal in Germany. The O&M-based results from the SOS server were available for parsing and display with a JavaScript-based webpage. Unfortunately, their implementation only returned real-time data when the page was refreshed by the user. This is because the O&M standard is very detailed and verbose, making it difficult to parse in real-time within a web browser. The architecture from the University of Bonn also did not explore publisher/subscriber systems for flexible client-side filtering of data and was limited to the specific traffic data stream available in their area, which they

admitted was not truly real-time (something common of public "real-time" traffic sources [6]).

2.3.2 Real-Time Web Architectures

A number of approaches have been taken by different research groups in order to build the infrastructure needed for real-time delivery of geographic data to the web, with special attention given to middleware that makes real-time communication possible. This section will look at the most significant such works, which include architectures based on publisher/subscriber technologies such as XMPP and JMS.

It is important to note that "real-time" in the context of the web has slightly different implications than in other areas of research (such as real-time operating systems where scheduling is a key factor). The "real-time" ability that applies to web applications, and therefore the one used in this thesis, is often called "near real-time". Accessible browser technologies such as JavaScript and Flash do not support threading, meaning that any new event on the client side must wait for the current function or method to complete execution before the event can be attended to. In addition, specialized techniques for dealing with the delay inherent of network-based communication systems (such as Quality of Service) are not assumed to be used since the architecture should emphasize ease of accessibility (for integrators as well as end users) in addition to supporting real-time data. As such, a "best-effort" approach is to be taken to minimize delays throughout the architecture.

2.3.2.1 Basic Publisher/Subscriber Architecture

Traditionally, the communication between two components of a software system involve one component that is programmed to send its message to another receiving component. In order to transfer the data from one sensor to *multiple* clients in real-time, a system is needed whereby each client can signal its own interest in receiving the specific sensor's data at runtime. The system must then ensure that all interested clients receive the updated sensor data the moment it becomes available to the sensor. The publisher/subscriber pattern (sometimes also called *publish/subscribe*, or simply *pub/sub*) is a well-established technique for delivering real-time data in this fashion.

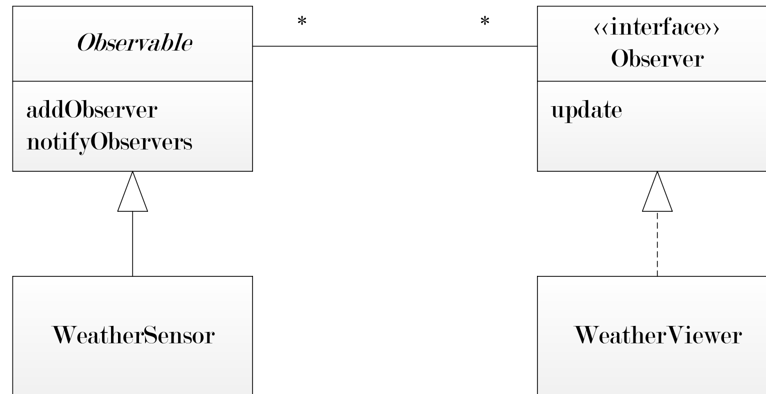

FIGURE 2.6: Observer Pattern for Sensor Data.

At the heart of a publisher/subscriber based architecture is the software design pattern known as the "Observer Pattern". This pattern was first introduced in the classic "Gang of Four" book by Gamma et al. [49] and was further described in [50]. Designed to reduce the amount of coupling between classes, the observer pattern uses an abstract class known as an *Observable* (sometimes also called a *Subject*). *Observable* has two methods: *notifyObservers* and *addObserver*. *Observable* stores a list of *Observer* instances. *Observer* is an interface containing an *update* method. Each *Observer* adds itself to the *Observable*'s list by calling *addObserver*. A *ConcreteObservable* that implements *Observable* is then able to use *notifyObservers* such that the *update* method of each and every *Observer* on the list is called. Using *WeatherSensor* as an example of a *ConcreteObservable* and *WeatherViewer* as an example of a class that implements the *Observer* interface, one can see how multiple viewers can instantly be updated whenever the weather sensor signals that it has new data by calling *notifyObservers*. Each weather sensor can therefore be monitored by multiple interested viewers, all of which will be notified in real-time when new data is available. In addition, the weather sensor does not need to know the number of viewers that will be interested in receiving its *update* messages, nor what they will do with this data. Adapted from [50], figure 2.6 provides the UML diagram for the observer pattern.

First introduced in 1987 by Birman and Joseph [51], the Publisher/Subscriber architecture builds on the observer pattern by applying it to distributed client and server systems. Data publishers (previously *Observable*) categorize their messages into different *topics*. They do not need to know how many subscribers there will be (if any) or what the subscribers will do with the received messages. Subscribers (previously *Observer*) signal their interest in specific message topics, generally

without knowing if there are any publishers for that topic. A message broker can be used to allow publishers to send messages on a specific topic and to allow subscribers to register to a specific topic. This way, publishers and subscribers can operate independently of one-another. This is different from traditional client/server communication, where the client is unable to send messages to the server if the server is not running, and the server is unable to receive messages from the client when the client is not running.

The publisher/subscriber architecture has been the foundation of most real-time communication research today for its ability to instantly *push* data to clients, rather than requiring clients to continuously poll for updates at regular intervals. This is especially important for GIS applications, where sensor networks can publish vast amounts of sensor data on predefined topics, and end-user client subscribers need only subscribe to the topics that contain the data they are interested in. The architecture proposed in this thesis must therefore build on the strengths of the basic publisher/subscriber architecture to allow for the real-time delivery of disparate sensor data to web-based clients and to make it available for collaborative use.

2.3.2.2 XMPP-Based Architecture

Work done at the University of Münster as part of the *52North* group [52] uses an architecture based on the Extensible Messaging and Presence Protocol (XMPP) to ensure the real-time distribution of alert messages. 52North implemented the non-finalized OGC standard known as Sensor Alert Service (SAS) [53], which specifies how to use the OGC SensorML format along with XMPP to deliver alert messages from sensor nodes to PC clients in real-time. Sensor nodes are able to register themselves with a SAS server and stream observation data to it through XMPP. The SAS server therefore acts as a registry for clients to know which sensor data is available. The SAS server is also used to define alert conditions by the client.

In this architecture, both Publisher (sensor node) and Subscriber (client) register themselves on the SAS server and communicate with each other using XMPP through a Multi-User Chat (MUC) channel [54]. XMPP is an open messaging standard that transfers messages by using XML and is managed by the Internet Engineering Task Force (IETF) [55]. Originally known as *Jabber*, XMPP allows

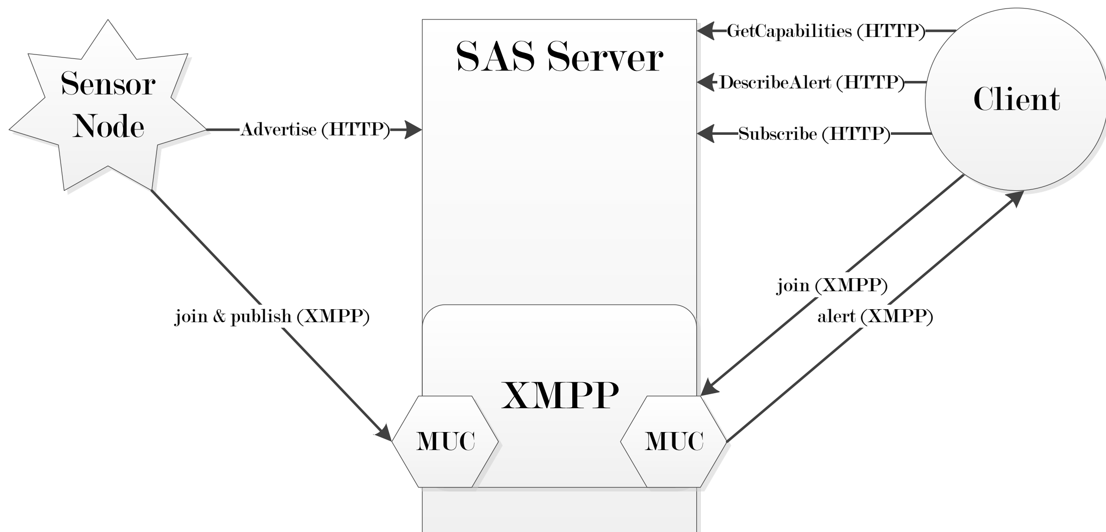

FIGURE 2.7: XMPP-Based Architecture for SAS.

for real-time extensible messaging and interoperability with other XMPP servers through a decentralized architecture. Since it was originally designed for instant messaging, XMPP also handles contact list maintenance and presence information (contacts can be online, away etc.). Custom functionality can be added on top of XMPP to extend its capabilities, as 52North has done by implementing the SAS standard and its sensor data oriented requirements.

The 52North architecture allows a sensor node to advertise its availability to the SAS server over HTTP. This includes the sensor's capabilities in the SensorML format, such as the type of data it is reporting, its reporting frequency, and its location. The SAS server then returns the address of a MUC channel for the sensor node to join (the *topic*), which the sensor node does by using XML messages as defined by the XMPP standard. The sensor node then begins publishing real-time data to the MUC using XMPP. A client can retrieve the list of sensor nodes and their capabilities from the SAS server and choose to subscribe to subscribe to a sensor node. When subscribing, the SAS server returns the address of the MUC where the sensor is publishing its data. The client is notified as soon as observational data is published to the MUC that meets the alert requirements defined in the SAS server. The general SAS architecture, as presented in [53], is shown in Figure 2.7.

52North has recognized the requirement for an architecture that includes push-based notification functionality to deliver real-time sensor data to a client. By

using an XMPP-based publisher/subscriber architecture, multiple clients can subscribe to be notified of the necessary sensor data. The 52North SAS implementation, however, is not designed for continuous real-time data streaming to web clients, but instead sends detailed information whenever an alert situation occurs. This means that the SensorML messages received by the client are large and verbose in nature since they contain the complete observation data for the alert event. Continuously streaming this amount of data and processing it within the web browser would be too computationally demanding for the average user's PCs. In addition, XMPP is not an ideal protocol for streaming to a web browser as it too requires a relatively expensive deserialization process. While Facebook Chat, Google Talk (commonly accessed through GMail) and Google Wave (discussed further in Section 2.3.3.1) have also relied on XMPP-based architectures to serve their large user bases, XMPP is generally limited to communication between servers, and a lighter format is delivered to the web client [56][57]. This is especially important if mobile device are to be supported, which are limited in computational power and memory. Google Engineer Dave Cridland expressed the following when discussing the Google Talk client for the Android mobile platform [58]:

> "GTalk API is not XMPP compliant, and will be less so going forward. The reason is that XMPP is too verbose and inefficient for mobile network connection, and the GTalk API will be moving to a binary encoding for the protocol between the client and the server."

As of November 2010, OGC is discontinuing development on the SAS standard and is instead working on a new standard for providing real-time access to sensor data and measurements known as Sensor Event Service (SES). The SES standard will continue to support XMPP but will also allow for other publisher/subscriber mechanisms to be used as long as the specified interfaces are implemented and the web-friendly Simple Object Access Protocol (SOAP) is used to communicate with the SES server [59].

2.3.2.3 JMS-Based Architecture

Another approach for distributing real-time data via the web was investigated by NASA scientists Tian et. al. [60]. Their objective was to implement a service-oriented architecture (SOA) based on the REST-style OGC Sensor Web Enablement (SWE) standards such as Sensor Observation Service (SOS, previously discussed in Section 2.3.1.2). OGC's web services and data encoding standards were used since the authors wished to seamlessly integrate a variety of disparate sensor data gathered by remote or *in situ* sensors and sensing agents while ensuring interoperability with various heterogeneous GIS components.

The communication middleware was identified by the authors as one of the most critical components of the architecture if data is to be transmitted in real-time. After considering remote procedure call (RPC) and object request broker (ORB) middleware alternatives, the authors found that "the data-centered, loosely-coupled and asynchronous nature of message-oriented middleware (MOM) makes it the perfect choice for sensor web services". They tested and surveyed several MOM alternatives before deciding to use a Java Message Service (JMS)-based MOM for its support of the publisher/subscriber messaging model and reliable, asynchronous message delivery.

JMS is a Java-based API defined by Sun Microsystems that allows for asynchronous messaging based on the publish/subscribe messaging software pattern. The purpose of a JMS Server (sometimes also called a *JMS Provider*) is to route messages between JMS Clients, which can be either JMS Publishers or JMS Subscribers. JMS Publishers publish messages to a certain "topic" on the JMS Server, and JMS Subscribers subscribe to that topic to asynchronously receive the messages. JMS Server components (of which there are several implementations, such as JBoss Messaging) can be deployed within application servers such as JBoss [61], while JMS Clients can be Java applications that use the JMS API.

Tian et. al. found that JMS served as a "perfect messaging substrate to construct a loosely-coupled and robust system to integrate distributed applications". The authors particularly liked that a message publisher (a component of the SOS Server) does not need to engage in a send-acknowledge-send cycle with a message subscriber (a component of the application server), and that the system cannot be tied down by a crashed subscriber. JMS was used to successfully publish sensor data from a SOS Server to a subscribing web client. The data for sixty temperature

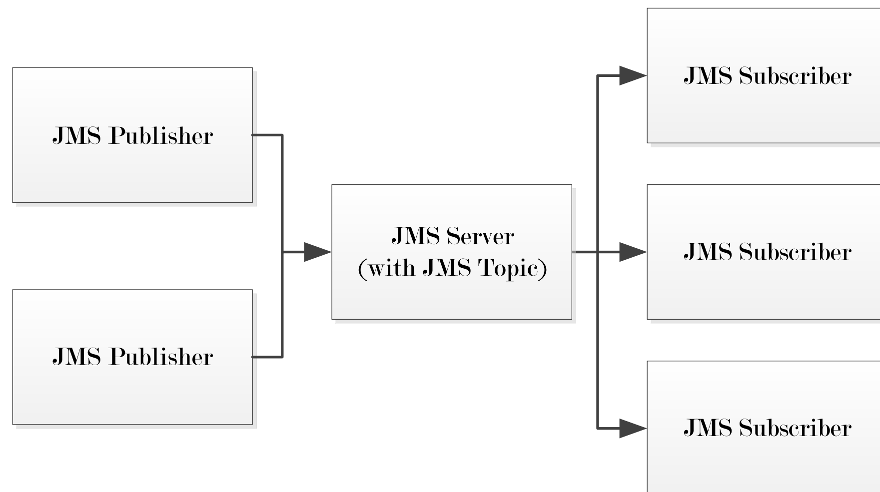

Figure 2.8: Publisher/Subscriber Messaging Using JMS.

sensors was generated by a sensor simulator and contained the sensor ID, latitude, longitude and temperature reading. The web client employed the Google Maps API and Asynchronous JavaScript and XML (AJAX) to fully take advantage of the asynchronous messaging characteristics of JMS and display temperature data within the browser at a rate of one sample per second. A simplified version of their architecture is summarized in Figure 2.9. A similar architecture for using JMS to deliver real-time data to a web browser was explored in [62].

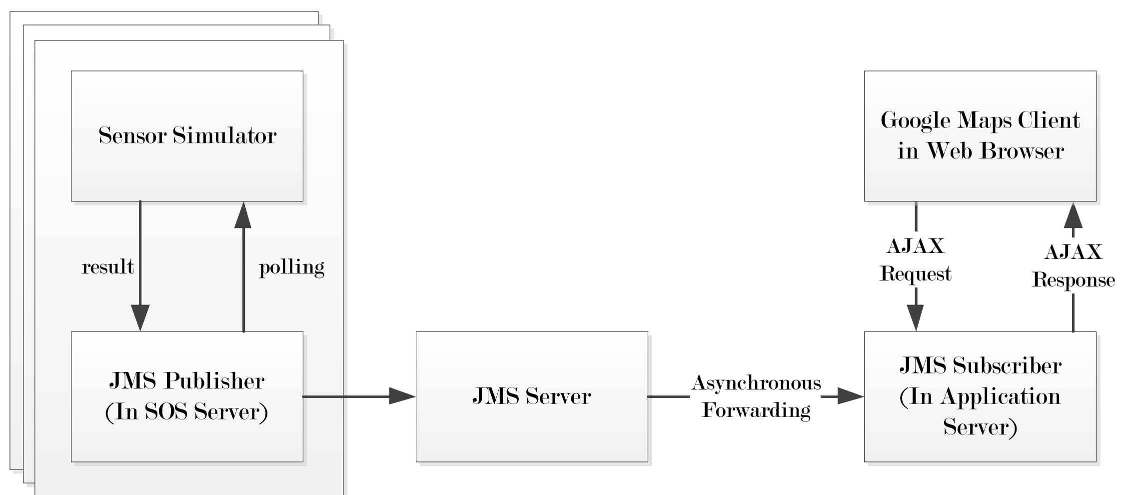

Figure 2.9: JMS-Based Architecture for SOS Data.

The research of Tian et. al. shows how the OGC SOS standard can be used to produce an open and interoperable architecture that, when combined with JMS-based publisher/subscriber messaging, allows simulated real-time sensor data to be displayed within a web browser using AJAX. This thesis, however, aims to go far beyond plotting simulated temperature data on a Google Map. In order to

truly prove that data is updating within a web browser in real-time, a source of data is needed that can be modified by the user to observe the delay until the update reaches the web browser. The use of video games as a real-time source of data was previously experimented with by Guido and Nanja [63], who used player location information from the game Unreal Tournament to observe the efficiency of different navigation systems in emergency response scenarios. Users should also be able to share and collaborate on the real-time data so that the data can be implemented for real-world uses such as decision making, education and public outreach.

2.3.3 Collaborative Platforms

As the amount of multimedia content on the web continues to grow at a staggering rate, users are increasingly looking for ways to instantly share it with colleagues and friends. Several commercial, open source and academic solutions have attempted to make it easier to share GIS content among many users in real-time by providing browser-based application platforms, as this section will show.

2.3.3.1 Google Wave

Perhaps one of the most sophisticated browser-based collaboration systems is Google Wave [64]. Created by the developers of Google Maps, Google Wave was first demonstrated in a developer preview at the Google I/O 2009 conference [65]. Visually similar to Google's GMail product, the left side of the user's screen contains a list of the latest updated "waves" while the right side contains the content of the waves. Unlike GMail, however, each wave can be shared with other users in such a way that all users see the same state of the wave in real-time. This means that, as one user types inside the wave, all other users watching the wave see the characters appear keystroke by keystroke. The developer preview showed five users co-authoring a wave in a fashion similar to Google Docs [66]. Users can also insert photos into waves to have them instantly become visible to their friends and co-workers invited to the wave. A "gadget" API exists whereby anyone can develop synchronized applications that can be inserted into waves to make the real-time sharing of a wide variety of content possible. In one sequence of the developer preview, a user was zooming, panning and drawing on a Google Maps

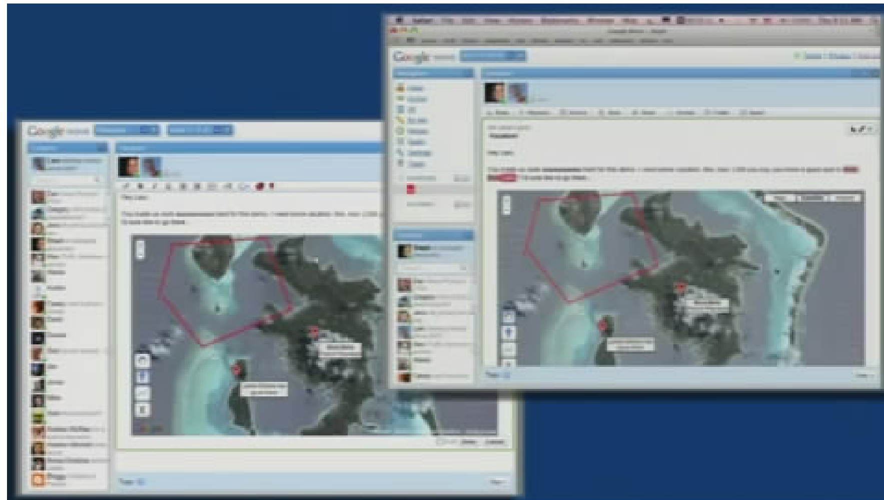

FIGURE 2.10: Collaborative Google Maps in Google Wave.

gadget to highlight a certain location while another user watched all those changes be reflected on their map in real-time. This is shown in Figure 2.10.

Google Wave uses the latest AJAX and HTML5 techniques to make this browser-based collaboration possible. In addition, Google developed several new Operational Transformation algorithms for conflict resolution of scenarios where many users are changing the same wave at the same time, as described by Google employee Abdessamad Imine in [67]. Google Wave also has an XMPP-based federation protocol for securely connecting users and waves distributed among disparate Google Wave servers [57]. The strengths and weaknesses of using XMPP for real-time messaging were discussed in Section 2.3.2.2.

One of the objectives of this thesis is to display real-time GIS data within a collaborative environment. While Google Wave has many strengths, it also has several key drawbacks that make it unsuitable as a collaborative environment for this thesis. One problem is the visual layout that squeezes the collaborative content into the right side of the screen and requires the user to scroll down within the wave to see its full content. This means that if a user inserted a collaborative map gadget near the bottom of a wave, other users in the collaborative session will not see the changes taking place unless they manually scroll to that section of the wave. All users may therefore not be seeing the same thing as they could be at different points in the wave, causing confusion in a collaborative setting.

Another problem with Google Wave is the overall complexity of using it for even

the most basic collaborative tasks. Instead of focusing on allowing users to complete a well-defined set of tasks, Google Wave's feature list is extensive and dynamic, with users even given the ability to develop and insert "robots" (in addition to gadgets) into waves to perform automated tasks on the wave content (for example, this allows text chat with real-time translation within a wave). Google Wave also included the ability to "rewind" a wave to see every keystroke that was made and by which user. Supporting this elaborate feature set as a web application required compromises to be made in important areas such as the user interface. Its reliance on incomplete HTML5 specifications, which the W3C describes as "too early to deploy" and "not ready for production" [68], means that some features are limited to Google's own Chrome browser and do not function in more widely adopted browsers such as Internet Explorer [69]. The combination of these and other factors resulted with low user adoption rates of Google Wave and ultimately led to the project's cancellation in August 2010 [70].

Nevertheless, Google Wave demonstrates many new concepts that are now possible by using the latest web-based technology to bring social real-time data to a web browser. Google recently released the source code and protocols for Google Wave to the open source community, and several open source projects have emerged to take the technology forward [71]. Although extensive modifications would be required to the server code and user interface, it is technically possible to adapt Google Wave to meet the collaborative real-time GIS platform requirements presented in this thesis.

2.3.3.2 Adobe Tour Tracker

Another solution for displaying real-time GIS data in a collaborative and social environment was shown by the Adobe Tour Tracker, which was developed by a team lead by Allan Padgett from Adobe [72]. Padgett provided cycling fans with a way for them to track the bikers on a real-time map during the Amgen Tour of California 2007, 2008, 2009 and 2010 (all of which were sponsored by Adobe). Adobe worked with the organizers of the event to have over 100 bikers carry GPS devices. These devices would use GPRS to publish GPS data to *GPS Listener* components on servers running Adobe Media Server software. Adobe's Real Time Messaging Protocol (RTMP) was then used to broadcast the real-time GPS information to subscribers using the Tour Tracker Flash plugin inside their

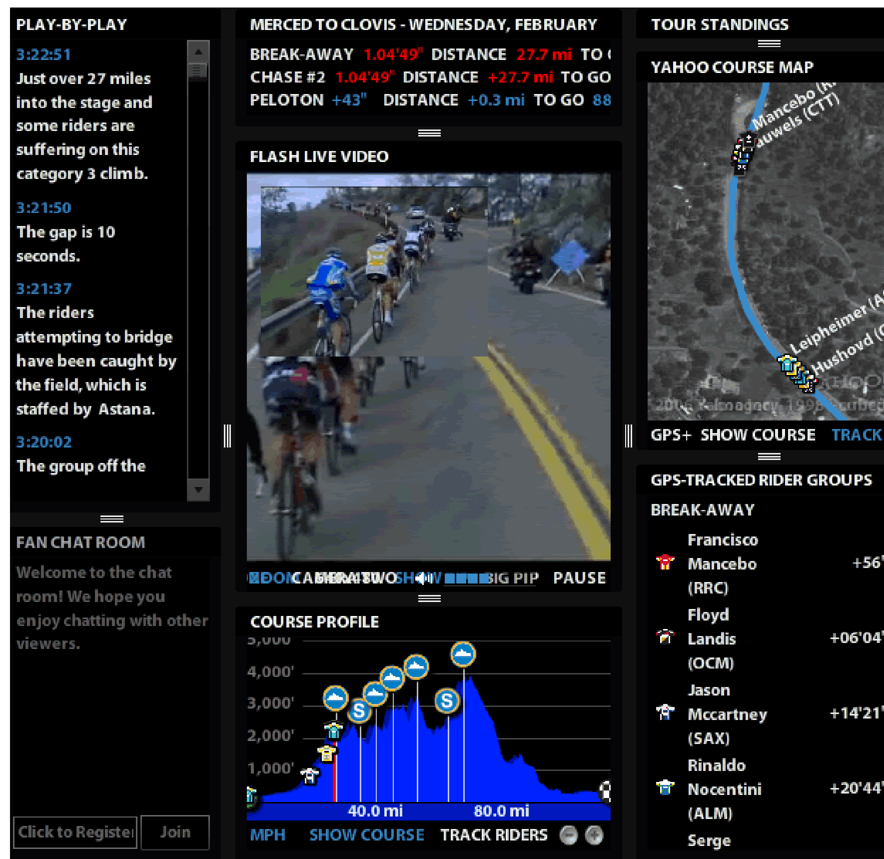

Figure 2.11: Real-Time Data in Adobe Tour Tracker 2009.

web browser [72]. Users were given a list of racers who they could subscribe to receive GPS data from. The Adobe Media Servers were clustered and clients were assigned to a random server to evenly distribute the load. Multiple live video streams from on-site cameras would also be available through the Flash Media Servers after being converted to a Flash video stream in real-time. In addition, a *Flickr Listener* component was added to the servers that would poll Flickr for the latest geo-tagged images from the event. A map would display the positions of the selected racers and the fan-generated Flickr images. Race statistics, audio commentary, live chat, and archived video clips were also available to enhance the user's experience of the event. The 2009 version of the Adobe Tour Tracker can be seen in Figure 2.11, with two video streams from the event displayed in the center and a map containing real-time GIS data displayed on the right side, among other race information.

The Adobe Tour Tracker is a good example of the interactive multimedia content that is now possible within a regular web browser. By using Adobe's Flash and Flex platforms along with the publisher/subscriber messaging systems provided

by the Adobe Media Server through RTMP, real-time GPS tracking was combined with live video, photos, and other race data into one web-based interface. While user interface modules can be resized, one drawback of the Adobe Tour Tracker is that the modules cannot be closed or moved, instead leaving users with Adobe's default information-heavy configuration as the only layout option. Another problem with Adobe's implementation of the Tour Tracker is that interacting with other fans of the event is limited to basic chat. It is not possible for one user to move a map to an interesting position and to instantly share this view with other users, for example. Users have to discover for themselves what the best video streams and Flickr photos are. The Adobe Tour Tracker's client and server code is not open source and new user interface modules can only be developed and released by Adobe. In addition, the application appears highly tuned for use with this specific cycling event and may perform inadequately if alternate data sources and formats are introduced. The Tour Tracker therefore makes for a poor collaborative platform in the context of this thesis.

The requirements and high-level design are presented in Chapter 3.

Chapter 3

High Level Design

This chapter presents the requirements and standard architecture that allows real-time sensor data to be delivered to a collaborative web-based environment. The standard nature of the architecture ensures that it can be applied to a wide variety of scenarios that require real-time geographic data to be delivered to a collaborative web application running within a web browser. In order to provide such reusability, the architecture will be presented in a platform-agnostic way and will therefore not specify any language of development or system on which to execute. As such, this chapter contains a very high level view of how the system is composed and how the various components interoperate. The functionality and responsibilities of all key components will be described. Their communication requirements to other components and their overall importance to the architecture will also be highlighted.

The architecture has its roots in some of the proven approaches and technologies presented in Chapter 2. In order to support a large variety of sensor data, a *Sensor Server* is required to accept real-time data from a *Sensor* source and to make it available for access from a web client. As a real-time architecture, the publisher/subscriber messaging model is required to allow each web-based user to act as a *Subscriber* to a stream of sensor data made available by at least one *Publisher*. Finally, the sensor data must be presented within a *GIS Web Application* built on top of a social *Collaborative Web Client Platform* that tracks online users and allows for real-time data to be shared between them. The final architecture features these and other components that, by working in unison, make it possible

to reliably and effectively deliver real-time GIS data to collaborative web-based clients.

3.1 Architecture Requirements

When building the architecture for any software system, one should always begin by establishing the requirements for the system. Once the requirements of the system are well understood, an architecture can be created to address them. The following requirements were determined for the architecture presented in this thesis. They are based on extensive domain research and the existing systems presented above.

Interoperable The components of the architecture should feature well-defined interfaces such that there is minimal coupling between components. Integrators of the architecture must therefore be given flexibility when deciding between component alternatives. Each component should perform only one role such that there is high cohesion. The architecture must therefore be capable of adapting to various existing GIS server and client standards and technologies.

Publisher/Subscriber-Based Messaging The components of the architecture must communicate by using the publisher/subscriber messaging model whenever possible, whereby components send data as soon as it is available. Publisher/subscriber messaging allows for high cohesion and low coupling between independently communicating components, while also allowing multiple recipients to receive the data of their choice from the available publishers. Finally, it adds fault tolerance to the system since the failure of one component does not affect the uptime of the entire system.

Real-Time Latency must be minimized throughout the architecture to ensure that users are receiving data that is as real-time as possible. This implies a careful selection of components and data formats to ensure that communication is light and reliable. In addition, the latency between collaborating users has to be very low to prevent discrepancies between the state of the application seen by the different users.

Accessible The architecture must be designed to allow access to a large audience with as little difficulty as possible, and must therefore not require exotic client components or technologies. The user interface must be familiar and easy to learn, while also ensuring the presence of the necessary information for decision making. Accessibility from mobile devices such as smartphones must also be supported, with special considerations given to user interface design. The scalability and reliability of the architecture must therefore be high, as this affects accessibility.

Collaborative The architecture must emphasize a highly social user experience within a synchronized collaborative environment. The environment has to provide a platform on top of which new synchronized applications can be developed and deployed. Through this capability, developers can extend the functionality of the environment by adding entirely new synchronized applications. This way, as new online services become available, the platform can be extended to support them and make their content collaborative. As a platform, it must provide easy-to-use APIs for developers. Additionally, the synchronization between users has to be done transparently such that developers do not have to worry about the necessary synchronization messages reaching all users of a session.

Based on Open Standards The architecture should make use of open sensor standards, communication standards, and web standards to ensure compatibility with components from a large variety of hardware and software providers. Open standards are generally developed and maintained through active collaboration between individuals from academia, industry, and government so that systems are not locked to one manufacturer's components. This vendor-neutral and platform-independent nature of open standards make them ideal for meeting the interoperability and accessibility requirements of the proposed architecture.

3.2 Component Design

One of the main objectives of the architecture presented in this thesis is to allow the interoperability of components created by separate entities. For example, the

architecture is to support a large variety of sensor data from disparate sensor networks, as well as a variety of client and server implementations. In order to achieve this, each component must have a well-defined role in the architecture. This means that components must exhibit high cohesion and provide unambiguous interfaces. While components must be able to communicate with each other, they must only be permitted to do so through these interfaces. Coupling between components must therefore be kept to a minimum. The publisher/subscriber pattern is one way of ensuring that messages can be delivered from suppliers to consumers without introducing excessive coupling between them since publishers do not need to know which specific subscribers, if any, are interested in their data [73].

The components of the architecture must also be flexible enough such that the architecture can be adapted to real-time sensor delivery scenarios. Whether end users need access to one important sensor or numerous large sensor networks, the architecture must support the removal and addition of components to account for such possibilities. The client's browser-based user interface should also be allowed to vary in complexity while still providing reliable and timely data. Deployment time must be minimized for integrators through the use of non-exotic technologies and open standards that help ensure compatibility with any existing components that integrators may have. Any additional software development and customization that may be required should also be facilitated by enforcing the use of well-defined development platforms. By making certain that such components are present, the architecture can provide significant flexibility for real-time GIS data distribution via the web.

3.3 Component Descriptions

The proposed architecture consists of eleven high level components. Each of these components have one clearly defined role and do not perform any function outside of this given role. The interfaces of the components define the ways by which their data or functionality is to be accessed. Some components are optional depending on the requirements of a specific deployment. A system may also contain more than one of some types of components. Together, these eleven components allow a variety of real-time sensor data sources to be delivered to a browser-based application for collaboration and analysis. The components of the architecture are:

GIS Web Application The GIS Web Application is an application, plugin or module built on top of the Collaborative Web Client Platform that provides the user with an intuitive interface to the sensor network data. Created by using the well-defined API of the platform, the application contains an interactive map that displays markers, dials and other information based on real-time data available through the platform. Rules are defined so that, for example, a certain range of values may indicate an "alert" scenario that should be highlighted using different colours.

The GIS Web Application also provides a mechanism to access a list of known sensor data. The list of known sensors is obtained by querying the Sensor Registry. The user "subscribes" to a stream of that sensor by selecting its name from a list. For example, a user may see a "Car" sensor on the list and choose to select it. While the user is subscribed, the car's latest real-time latitude and longitude values will be available to the GIS Web Application and can be animated on the map as a moving icon.

A search box is another essential component of the GIS Web Application. Once a search term is entered into the box and the "Search" button is pressed, a geocoding service is used to resolve the search string to a latitude/longitude value. The map is then centered on that location. For example, if the user enters "Ottawa", the map will center on a latitude value of 45.393348 degrees and a longitude value of -75.695610 degrees and at a zoom level appropriate for cities. Such functionality will be familiar to users of popular mapping websites such as Google Maps.

The GIS Web Application must also make at least one "base layer" available to the user. The base layer is made up of the actual map tiles that the user sees and may originate from a variety of external sources. The base layer tiles are served using an interactive "slippy map" interface to increase usability. Finally, use cases such as panning the map, zooming the map, switching the base layer, or loading a specific sensor are synchronized with all other users that have been defined by the Collaborative Web Client Platform to make up a collaborative session.

Collaborative Web Client Platform The Collaborative Web Client Platform is responsible for containing the GIS Web Application and providing the necessary functionality to make the GIS Web Application collaborative and easy to access for GIS-related tasks. As a "collaborative platform", its goal

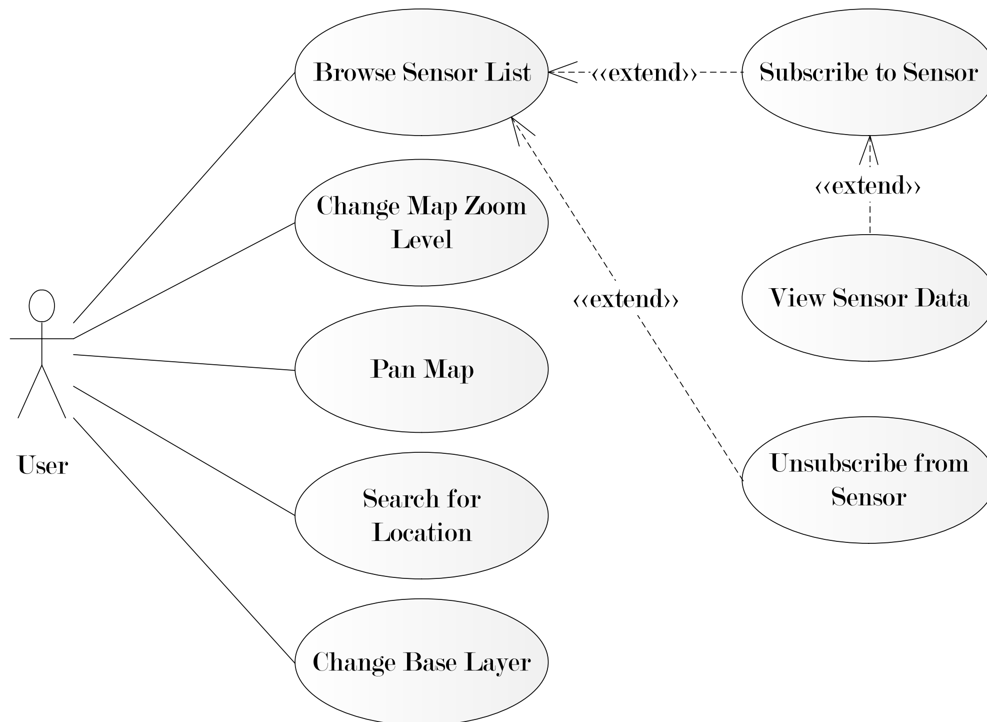

FIGURE 3.1: GIS Web Application Use Case Diagram.

is to allow various new synchronized applications to be developed and deployed by using its well-documented APIs. Developers should not have to worry about synchronization messages reaching the appropriate users when developing new applications on top of the platform. The process of developing collaborative applications should not be much different than for single user applications. Xu, Rose and Lin [74] developed a multi-layered architecture where a collaboration layer is defined to handle communication. It used an Events Listener Module such that all that is required of the application developer is to fire events to be captured by a collaboration manager. This approach would allow different GIS Web Applications (among many other types of applications) to be developed for the platform with relative ease, whereby the applications developed are affected by the real-time data from GIS sensors as well as from the real-time actions of other users.

The Collaborative Web Client Platform is also responsible for using the Application Server to track the online status (also called "presence") of users, as well as each user's list of contacts and the presence of those contacts. As such, it must first authenticate the user by requesting their username and password. Once the user has been authenticated, the platform provides the social invitation mechanisms required for users to establish a multi-user

"collaborative session". The state of any application within a collaborative session is synchronized in real-time among all users of that session. This approach differs from solutions that rely on screen-sharing techniques, where a user decides to share their desktop view with other users [13]. Since the screen-sharing technique is essentially taking multiple screenshots of the user's desktop per second and streaming them to the other users, it requires all users to have a large amount of bandwidth, especially when services such as video chat are also present. Collaborating on video data in the Collaborative Web Client Platform means that the video is rendered on each user's local machine within their own instance of that application. Videos are not re-encoded as part of a remote screen update; they therefore run at full speed for all users and synchronization is ensured through event-based signals.

The Collaborative Web Client Platform also defines how its applications are organized and displayed, such as whether it's within static or dynamic portions of the screen. More advanced platforms aim to recreate the familiar desktop experience by organizing applications into windows, offering flexible layouts for multiple application instances. This is particularly useful when performing advanced analysis of GIS data from multiple sources while collaborating with several others. Such a platform is often called a "WebOS" as it exhibits many of the usability features of a regular operating system while remaining accessible from a regular web browser. Finally, a Collaborative Web Client Platform provides a default way for users to communicate, such as through text or video chat. Figure 3.2 shows what the Collaborative Web Client Platform provides from the user's point-of-view.

Application Server The main function of the Application Server is to make the Collaborative Web Client Platform (and therefore the GIS Web Application) and its collaboration features available to users from within a web browser. The Application Server handles the requests from the Collaborative Web Client Platform and accesses the necessary services to fulfill the requests. This includes obtaining a list of active sensors from the Sensor Registry, as well as managing Subscribers to the requested sensors and receiving their real-time data. When subscribing to a sensor, the Application Server is also responsible for notifying the Smart Publisher to begin receiving and publishing data from external sources. When unsubscribing, the Application Server must notify the Smart Publisher to stop requesting data for that sensor.

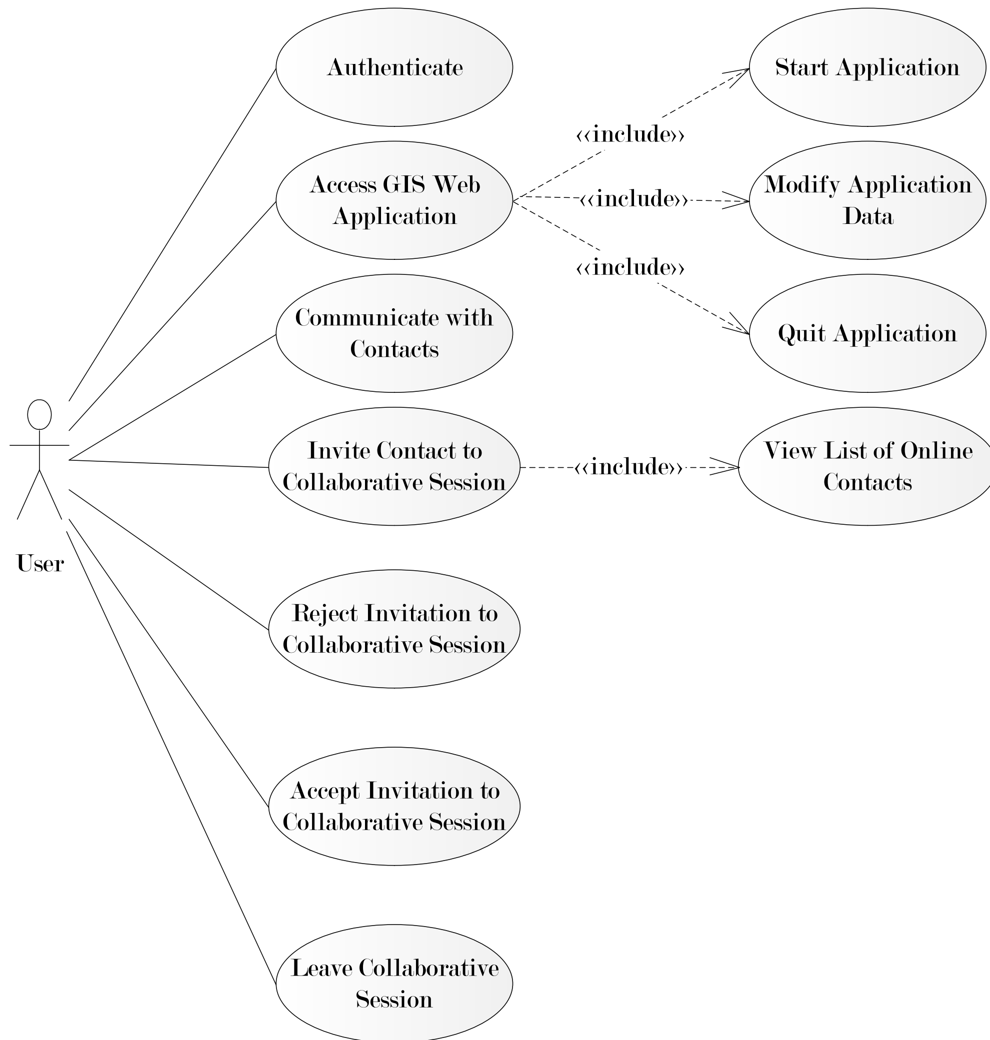

Figure 3.2: Collaborative Web Client Platform Use Case Diagram.

The Application Server must be stable, reliable, secure and always accessible. Initially, a user's web browser will request the Collaborative Web Client Platform from the web server within the Application Server. Once the client platform is loaded in the user's web-browser, the Application Server takes note that a new user is online. The user's client platform is then sent information such as which other users are currently online and available to collaborate with. When the user loads an application within a collaborative session, the Application Server must provide the synchronization mechanisms to ensure the correct messages reach the correct users. It must therefore track users who are coming online, going offline, and joining or leaving different collaborative sessions (including users who join a session that was previously started by other users).

The Application Server may also provide support for a large number of users

per session through a cloud-based architecture. Other advanced features include the ability to access data, and even entire applications, from users on different deployments (or *domains*). In addition, the application server may make use of a database to manage user and application state information.

Subscriber The Subscriber is a simple application that connects to a messaging server supporting the publisher/subscriber messaging paradigm. It provides methods for the Application Server to use to subscribe or unsubscribe to various sensors requested by the client side. Real-time sensor data is "pushed" to the Subscriber by the Publisher/Subscriber Server based on the "topic" which the Subscriber has registered a subscription to. This decoupling allows multiple subscriber instances to operate independently of the number of publishers present in the system.

Publisher/Subscriber Server The Publisher/Subscriber Server is a standards-based messaging server which routes messages between the Smart Publisher and the Subscriber. Each Smart Publisher sends messages to the Publisher/Subscriber Server as they become available from the Sensor Server. As requested by the Smart Publisher, all messages are assigned a certain topic which is managed by the Publisher/Subscriber Server. The Subscriber then subscribes to a topic to be notified of new data in real-time. In order to ensure the prompt delivery of sensor data, it is important that the Publisher/Subscriber Server offers reliability and high real-time performance.

Smart Publisher The Smart Publisher is, at its core, a Publisher application that flags sensor data with a topic and sends the sensor data updates to the Publisher/Subscriber Server in real-time. Multiple Publishers can be transmitting data from multiple sensor sources to the Publisher/Subscriber Server at the same time, and publishers do not know how many subscribers (if any) have registered to receive their data.

In this architecture, the Publisher is given additional intelligence for requesting real-time sensor data from the Sensor Server based on a list of sensors requested by the GIS Web Application through the Application Server. The Application Server must notify the Smart Publisher of any new subscriptions or unsubscriptions so that the Smart Publisher may schedule its requests to the Sensor Server accordingly. Both the request to the Sensor Server and the response from the Sensor Server must be completed using a common

GIS data standard and must be in a format that is optimal for real-time transmission. In addition, the data received from the Sensor Server should be converted to a format that is as light as possible before it is sent to the Publisher/Subscriber server.

Figure 3.3 shows the typical sequence of events required to deliver real-time data to the client side. The User sends a *Subscribe Request* from the GIS Web Application and through the Collaborative Web Client Platform. This request reaches the Application Server, which sets up the Smart Publisher to begin publishing data and instructs the Subscriber to subscribe to a sensor via the Publisher/Subscriber Server. As the sensor data is received by the Smart Publisher, it is converted and asynchronously forwarded to the User through the Publisher/Subscriber Server.

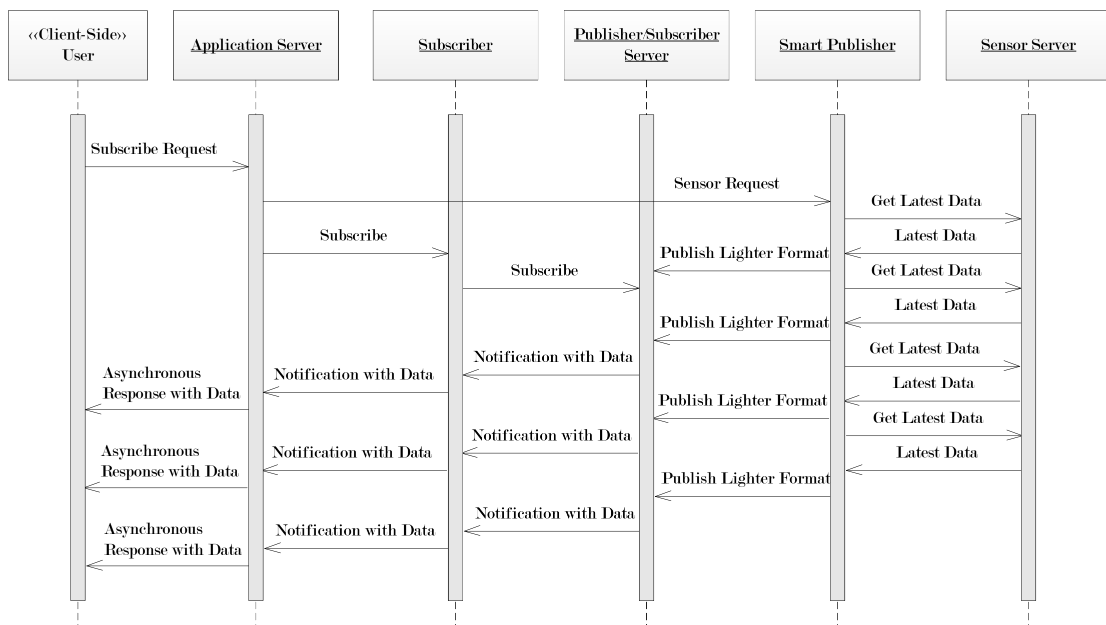

Figure 3.3: High Level Publisher/Subscriber Message Sequence Chart.

Sensor Server & Spatial Database The Sensor Server wraps the sensor data sent by the Sensor Feeder using a well-defined sensor services standard which can be polled by the Smart Publisher. Once converted to a lighter format, the Smart Publisher transmits the data to the GIS Web Application where it can be parsed and displayed to the user. It is important that the sensor standard selected for the Sensor Server is open such that other Sensor Servers may easily be integrated into the architecture. The Sensor Server may also contain a Spatial Database component for archiving GIS data. This data

can be accessed at any time from the client side for generating time-based graphs or other visualizations that do not require real-time data.

Sensor Feeder The Sensor Feeder is responsible for polling the Sensor as frequently as possible for updated data. The data obtained from the Sensor is then sent to Sensor Server where it can be hosted in a standards-compliant fashion for access from the client. The Sensor Feeder must therefore submit the real-time data to the Sensor Server in a way that is compliant with the GIS standard of the Sensor Server. Since there exist a large variety of possible Sensor types, the Sensor Feeder must also be able to communicate with the Sensor to properly retrieve the available data.

Sensor The Sensor is the physical device which contains the latest real-time result (typically a measurement value) about the property it is observing. For example, a sensor may return a result of “-2℃” for the property of “temperature”. The sensor does not process the gathered data, but rather only makes it available to other components. A simple example is a USB weather sensor which provides an API through which the Sensor Feeder can access its latest raw data. Sensors may be connected to other Sensors as part of a Sensor Network.

Sensor Network The Sensor Network component is a sensor indexing service containing the locations of various Sensors and the Sensor Servers to which the Sensors belong (if any). Sensor Networks are discovered by the Sensor Registry through different searching techniques so that their Sensors can be made available for subscription from the GIS Web Application. Sensor Networks may contain Sensors that provide data in various formats, some of which may need to be adapted to the known Sensor Server formats by using an appropriate Sensor Feeder.

Sensor Registry The Sensor Registry uses various external databases, searching techniques and other methods to discover existing Sensor Networks. Once the Sensor Servers of a Sensor Network are discovered, the Sensor Registry requests the available Sensors and sensor data from the Sensor Server. The Sensor Registry then makes the list of known sensors and measurements available to the Application Server as a series of services. The Application Server sends the list of Sensors to the GIS Web Application via the Collaborative Web Client Platform such that users can browse all available sensors.

Users can then choose to subscribe to the items of interest such that the Smart Publisher can begin publishing their data. In this architecture, the Sensor Registry is a centralized component. The message sequence chart in Figure 3.4 shows how the Sensor Registry uses a Sensor Network to discover new Sensors and make them available to the client side (which consists of the Collaborative Web Client Platform and the GIS Web Application) for subscription.

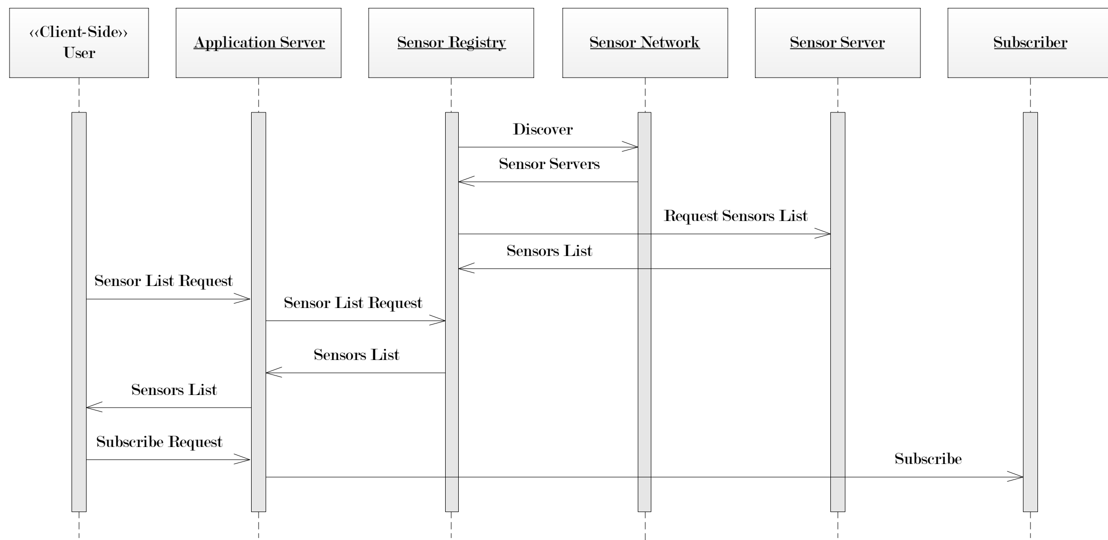

FIGURE 3.4: High Level Registry Message Sequence Chart.

Figure 3.5 shows the final high level architecture based on the eleven components described. While the architecture defines eleven components, the sensor network and sensor registry components can be omitted in smaller implementations consisting of a pre-defined list of sensors. The items shown within a cloud graphic can be located in a different domain and there typically exist multiple instances of each of these components.

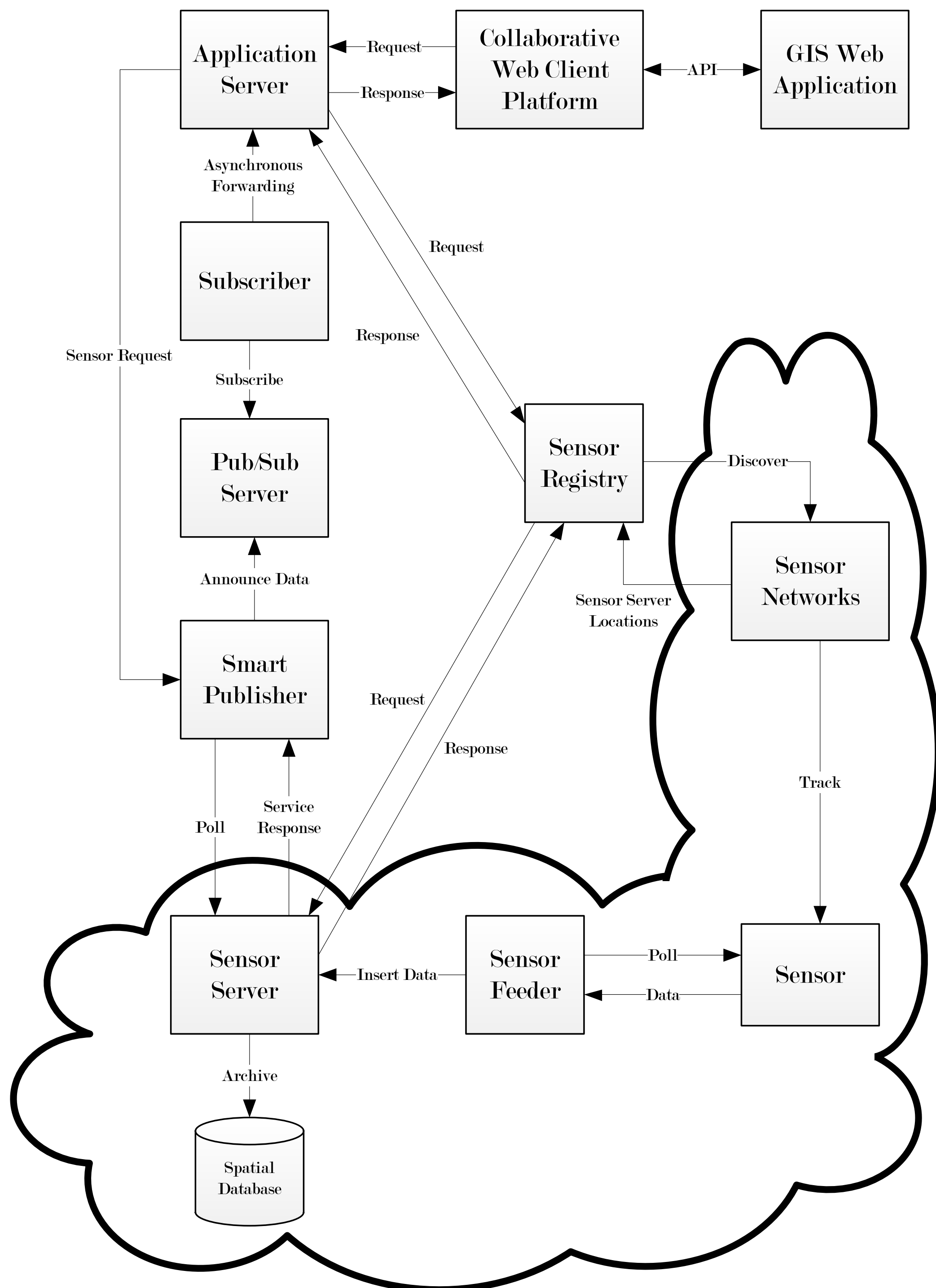

Figure 3.5: Real-Time Web-Based GIS Architecture.

Chapter 4

Low Level Design

This chapter presents the low level design and implementation details that make it possible to deliver real-time sensor data to collaborative users within a web browser. The language of development, system on which to execute, and communication methods between components will be given in the context of the architecture defined in Chapter 3. This chapter will therefore provide a low level look at how each of the components is built and how open standards are used throughout the architecture.

In order to best demonstrate the viability of the proposed architecture, two web-based platforms have been extended to include support for GIS and sensor networks. One is based on JavaScript/AJAX (UC-IC), the other on Adobe Flash (Watch Together). These two Collaborative Web Client Platforms were used since their real-time nature is ideal for map-based applications dealing with sensor data, and both platforms had collaborative GIS Web Applications built on top of their APIs. The server-side technology used to implement the majority of the architecture is based on Java, and especially the Java Platform Enterprise Edition (JEE). The publisher/subscriber messaging defined by the architecture is done by using the Java Message Service (JMS) standard from Sun Microsystems.

One of the key requirements of the real-time delivery architecture proposed in this thesis is that it be interoperable with components from various vendors. The implementation must therefore be free of proprietary technologies such that the solution can be applied to a variety of hardware and software systems from various vendors. It should therefore not matter if an application server is a JBoss Application Server, a Sun Glassfish Enterprise Server, or even a Flash Media Server.

The interfaces that provide data to these servers should be the same. The Sensor Observation Service (SOS) standard allows a wide variety of sensor information to be packaged in a way appropriate for web-based delivery, but other GIS standards exist (as shown in Section 2.3.1) or may emerge in the future.

4.1 A Real-Time Platform for Multi-Domain Collaboration

A real-time platform that supports collaboration across multiple domains has been developed and is known as UC-IC. UC-IC is a cloud-based real-time collaboration (RTC) platform which allows users to send, share and distribute control over applications of any nature in real-time. The Session Initiation Protocol (SIP) standard is used and extended to bridge users on different domains such that they can discover other users and collaborate on applications that were previously inaccessible. Users can concurrently collaborate on multiple applications from several domains, where each application can be part of multiple collaborative sessions. This section describes the server-side and client-side design of UC-IC before presenting how a real-time GIS web application was implemented to take advantage of UC-IC's collaborative nature.

UC-IC was developed in collaboration with Robin Tropper, Rabih Dagher, Bogdan Solomon, Bogdan Ionescu and Dr. Dan Ionescu of the University of Ottawa's NCCT Lab. UC-IC is described extensively in the Master's Thesis of Robin Tropper entitled "New Architecture and Programming Paradigms for Cloud Collaborative RIA" [75].

4.1.1 Application Server

JBoss was selected as the application server to power the UC-IC Server for its proven and open source implementation of JEE technologies [61]. Figure 4.1 and Figure 4.2 show how the UC-IC server side is divided into two main subsystems: *PersistenceSubsystem* and *ClientDisplaySubsystem*. The *EJB Container*, along with a *Database*, are used for storing user and application information across sessions as part of the *PersistenceSubsystem*. The *ClientDisplaySubsystem* includes

a *Servlet Container* for implementing the collaborative functionality and generating the HTML/JavaScript output for the user's browser. Besides regular *Users*, the system identifies *Administrators*, which can configure per-user access control for specific UC-IC applications within a domain (among other settings). Each instance of the UC-IC Server running within a JBoss deployment is considered to make up a *domain*. The server is designed for independent deployment by multiple domains on the cloud and allows users from one domain to collaborate with users from other domains.

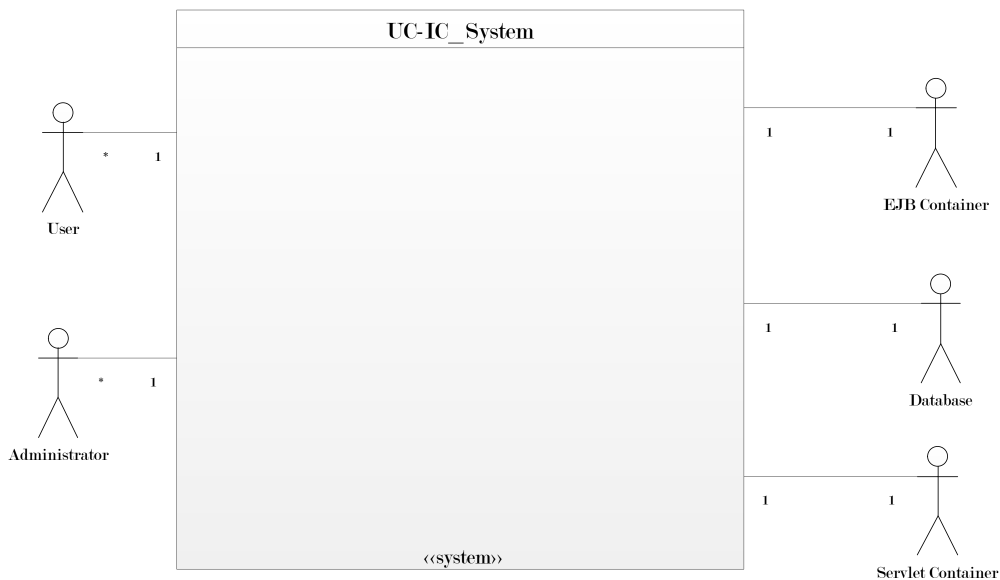

Figure 4.1: UC-IC System View.

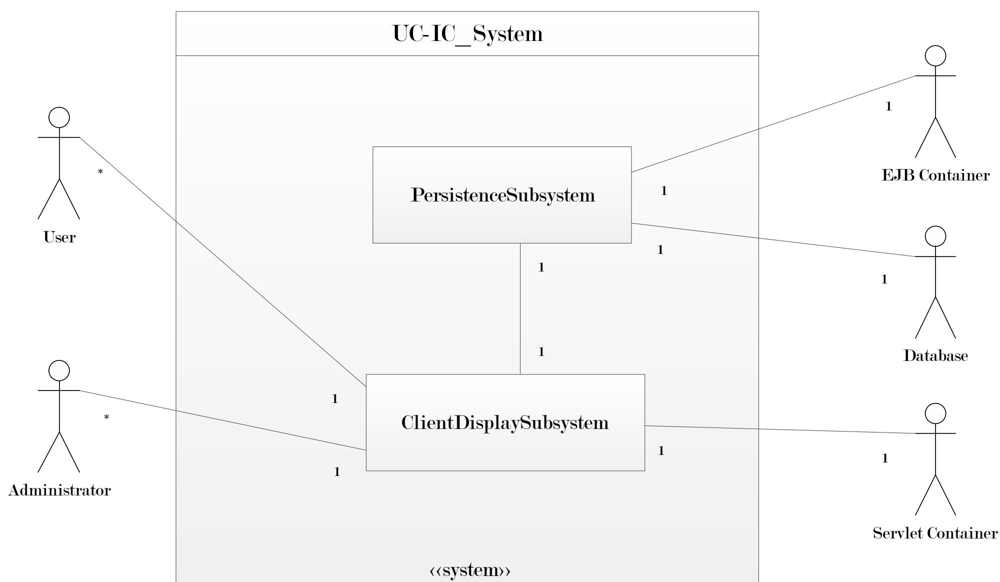

Figure 4.2: UC-IC Subsystem View.

Figure 4.3 shows the basic server-side components of the UC-IC Server. The *DisplayController* receives requests from web clients. The web browser's User Agent is identified using the *DeviceDetector* component so that user interface customizations can be made for smartphone browsers. The *DisplayController* then forwards the user's requests to the *BusinessLogic* component. The *BusinessLogic* component is the core component of the server and communicates with the *UC-ICApplication* component to retrieve the application content (such as for the GIS Web Application). The *BusinessLogic* component also communicates with the *Collaboration* component, which is responsible for managing the collaborative sessions and using the *SIPStack* component to communicate with the SIP Proxy server. SIP is used to allow the connection of any user to multiple collaborative sessions and to track user presence across various UC-IC deployments. The *BusinessLogic* component also communicates with the *StatefulSessionBeans* components of the *PersistenceSubsystem*. *EntityBeans*, which define the data that is to be persisted for returning users, are written to a database using JDBC and the Hibernate component of JBoss.

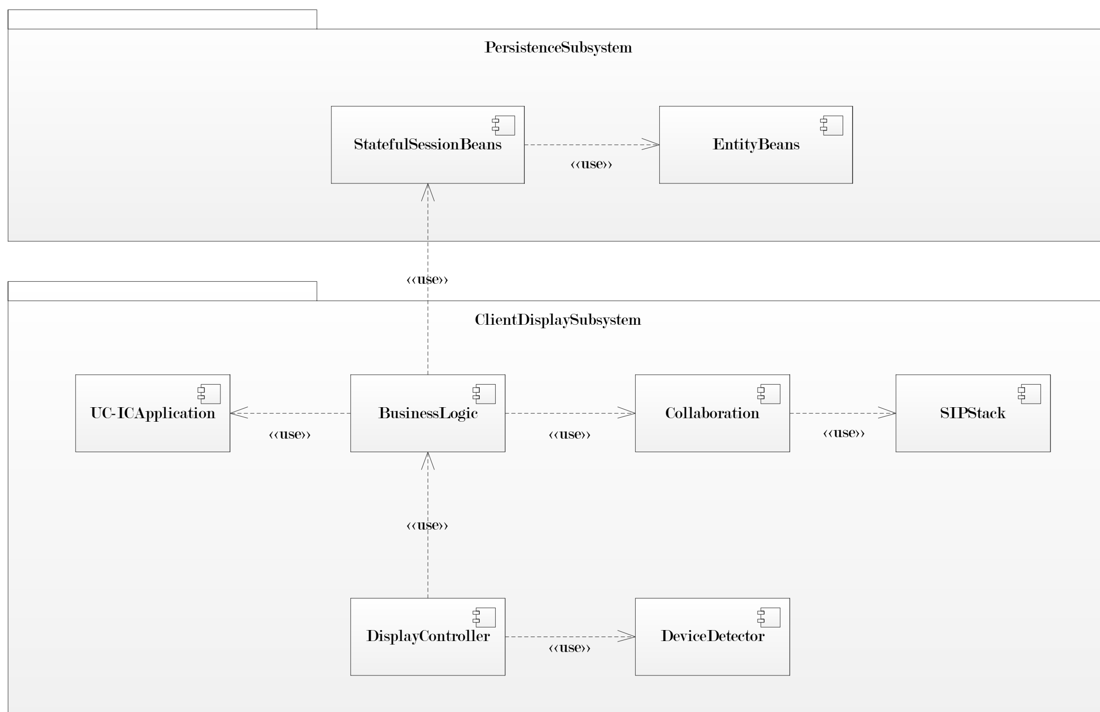

FIGURE 4.3: UC-IC Server-Side Component Diagram.

The most significant part of the UC-IC Server is the *Collaboration* component; its class diagram is shown in Figure 4.4. The *Collaboration* component consists of the *CollaborationManager*, which is responsible for creating the sessions and populating them with users and applications. Through the *SIPManager* class, it

also instructs the SIP Stack to send the appropriate SIP requests to the interested parties, as well as to receive SIP requests from other entities. The *CollaborationManager* class uses the *SensorService* class to subscribe to sensors via the Publisher/Subscriber Server of the architecture based on the sensor IDs, locations and observed properties (temperature etc.) requested by a client. The *SensorService* class also notifies the Smart Publisher of the subscriptions and then receives asynchronous notifications whenever there is a sensor update (as described in Section 4.3).

The *CollaborationSession* class of the UC-IC Server holds information about the collaborative session, along with a list of users and applications. The *CollaborationApplication* class contains information about the applications currently running in a collaborative session, so that when a new user is invited, those applications are pushed to that user. The *CollaborationUser* class contains information about the user, such as their SIP address and permissions. Finally, the *CollaborationPermission* class is used by administrators to give the user certain privileges such as adding or removing applications, inviting or removing users, and changing the permissions of other users.

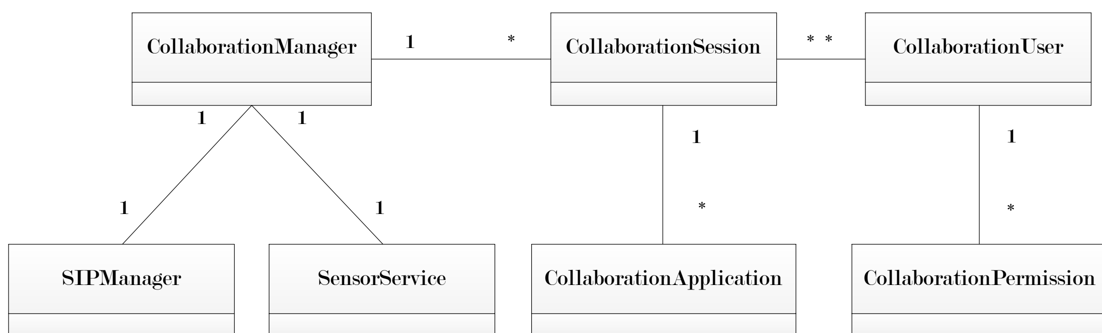

FIGURE 4.4: UC-IC Collaboration Class Diagram.

The UC-IC Server includes the ability to bridge servers from different domains. This is achieved by using the Session Initiation Protocol (SIP) as the connection protocol between the different servers, allowing users from different domains to be invited to - and thus participate in - the same collaboration session. Features such as user presence and discovery, user invitation, as well as session setup and teardown are thus managed in a multiple domain setting.

SIP is an application-layer control (signaling) protocol for creating, modifying, and terminating sessions with one or more participants. These sessions include

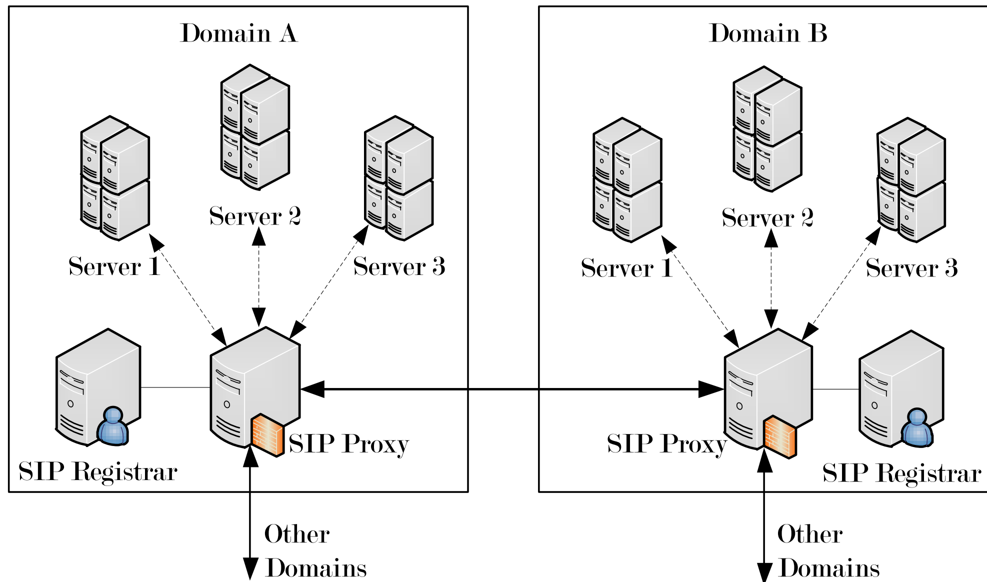

Figure 4.5: Two UC-IC Domains Communicating Through SIP.

Internet-based telephone calls, multimedia distribution, and multimedia conferences. The specification includes requests for the registration and invitation of users, as well as the setup and teardown of sessions [76]. Incorporating SIP into the architecture allows collaboration sessions to span multiple domains, thus giving each user the power to offer services (in the form of applications) that were not previously accessible to other users.

Figure 4.5 shows an example with two clusters of servers, each representing a different domain consisting of three servers. The domains are connected together through the SIP server of each of the clusters. Each distinct server may contain different applications and data that will instantly become available to other users invited to a collaborative session or who were sent an application window. Invitations can thus be sent out to users no matter what domain they are in, and sessions are established across multiple domains with the correct messages reaching users wherever they are located.

4.1.2 Collaborative Web Client Platform

The UC-IC Collaborative Web Client Platform is a web-based implementation of the desktop metaphor, consisting of familiar windows, icons and menus. The platform allows any window to be "sent" to another user. For the user, this is done simply by dragging a window onto an icon on the web-based desktop

representing the user who is to receive the application (and who must acknowledge a dialog to accept it). Users on the list are dynamically added and removed based on connections to the UC-IC Server (including from other domains via SIP). The application sending process is unique in that the entire application logic, in addition to the current data within that application, is sent as part of the window. Sending a map application, for example, means that the receiving user not only receives the application logic (the user interface and geographical functions called by the buttons), but also the current data and state of the application (the exact position where the map is centered).

It is important to note that this concept of "sending" a window does not necessarily mean that the user doing the sending no longer retains the application. Rather, by dragging the user icon to the application window, both users can have the same window open as part of a collaborative session. The inherent collaboration built into UC-IC ensures that any actions performed within that application are automatically synchronized to the other user. For example, any panning or zooming of a map will be communicated instantly to the other user's browser so that, as much as possible, both users always see the same application state. The environment was named "UC-IC" to highlight these collaborative characteristics.

Real-time collaboration on UC-IC is supported for applications programmed in, or able to communicate through, DHTML (AJAX). This makes it easy to develop new collaborative applications for the environment since this includes a variety of web-based technologies, including JavaScript, Adobe Flash and Java. Other existing applications built on the UC-IC platform include a text chat application, a rich-text editor for live co-authoring (similar to Google Docs [66]), a collaborative video player, a synchronized photo viewer, a web browser, a calculator, and a drawing application that can be reused to add annotations on top of other UC-IC applications.

Real-time collaboration updates and sensor updates within the browser are made possible by an advanced XML-based syntax and a dynamic resource loading technique known as "AJAX-Push" [77][21]. The web browser is able to send AJAX messages to the UC-IC Server, which directly invokes JavaScript object methods on other users' workspaces. To do this, applications must notify a client-side collaboration manager which prepares browser-to-browser RPC messages. These messages will replicate the user's actions on the corresponding applications of the other users participating in the same collaborative sessions.

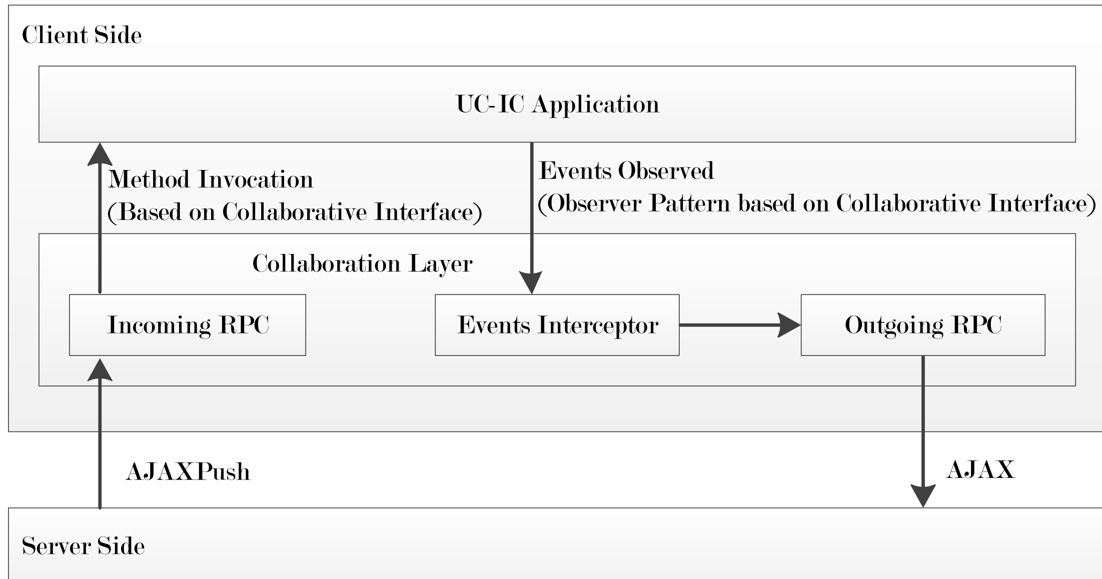

Figure 4.6: UC-IC Client Platform Components.

In contrast with mainstream Web 2.0, the client is fully responsible for creating and modifying its user interface, including the creation and destruction of user applications. The UC-IC Server only loads required resources, provides data and manages communications. Rather than simply accepting and inserting DHTML content fed by the server, the client handles user actions entirely, as well as provides full management for incoming and outgoing messages which include the replication of actions that originally occurred on the workspaces of other users.

A layered platform for collaboration is thus provided where applications designed for single use can participate in cloud collaboration with little modification. This platform, shown in Figure 4.6, follows a model similar to [74], whereby the *Observer Pattern* is used to capture events that need to be replicated on the workspaces of multiple other participating users. Developers of new applications need only define a "collaborative interface" (containing the callback functions, required arguments and their data types) to ensure that notifications are serialized and transmitted for the actions of their choice. The *Collaboration Layer* contains an *Events Interceptor* which observes all applications in each session so that the correct observers (users) can be notified. By using the defined *collaborative interface*, the Collaboration Layer automates the XML-based serialization and deserialization of browser-to-browser remote procedure calls which are broadcast to all users participating in the RTC session. The broadcast, as well as all required controls for data integrity and permissions, are ensured by the UC-IC Server.

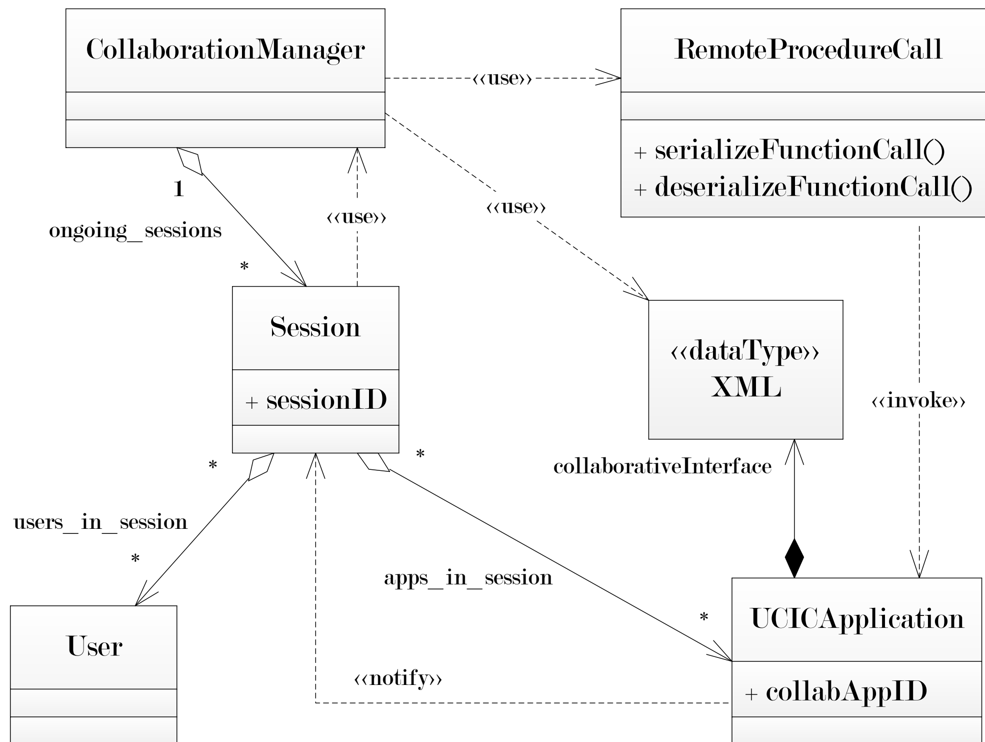

FIGURE 4.7: UC-IC Client Side Class Diagram.

The UC-IC client ensures real-time communication by encapsulating the collaboration information for sending via the UC-IC Server. A basic class diagram is shown in Figure 4.7. The client-side *CollaborationManager* manages any number of *Sessions*, each of which connect any number of *UCICApplication* instances with any number of *Users*. Because the users, applications, and sessions are abstract entities managed by the UC-IC Server, their location is unimportant; this makes it possible to connect users and applications from different domains. The mapping that determines which applications participate in which sessions is obtained by using two identifiers: a *sessionID*, which belongs to the *Session* object such that the server can find users in the same RTC session, and a *collabAppID*, which is assigned to the application to uniquely identify it on the workspaces of all participating users' environments. By combining *sessionID* and *collabAppID*, a key is obtained by which a given application can concurrently participate in different collaboration sessions. The collaboration manager also uses the application's *collaborative interface* to serialize and deserialize an XML-based *RemoteProcedureCall* which replicates the action directly on the components concerned for all participating users.

To implement a new application in UC-IC, a JavaScript class needs to be defined

that creates a new instance of the *UCICApplication* class. Since JavaScript lacks the formal concept of an interface (among other features typical of object oriented programming languages) [78], *UCICApplication* is used as an interface indicating the functions that implementing classes are to override, as well as properties that can be defined. *UCICApplication* provides many properties such as *frameTitle*, which is used to define the title that is to appear in the application's window, and *applicationDefinitionString*, which is a string containing the HTML content to be displayed within the window. This HTML content can then access other functions within the same JavaScript class or within other objects. In addition, the functions *getXmlData()* and *restoreFromXml()* are overridden to define the application's collaborative interface, which is done by building an XML string containing the variables that should be stored when the application is sent to another user (and is also called for event-based synchronizations). Finally, a *destructor()* function must be implemented to clean up any JavaScript objects created and prevent memory leaks within the web browser.

Since UC-IC recreates many aspects of a typical operating system, it is an elaborate project consisting of thousands of lines of code spread over nearly one hundred classes. A thorough description of its many features and its complete developer API can be found in "New Architecture and Programming Paradigms for Cloud Collaborative RIA" [75].

4.1.3 GIS Web Application

The GIS Web Application implemented on the UC-IC platform was named Joint Intelligence Picture (JIP). It consists of four separate UC-IC applications, where each application appears as a separate UC-IC window. All four applications were implemented by extending the *UCICApplication* class. The final classes are shown in Figure 4.8.

JIP Search Window Implemented as a class named *JIPSearch*, the *JIP Search Window* contains a search box and "Search" button. Clicking the button passes the string from the text box to the *JIPMap* class by calling the *loadMap* function.

JIP Map Window The *JIP Map Window* contains an interactive "slippy map" on which the real-time sensor data is made available. The Google Maps

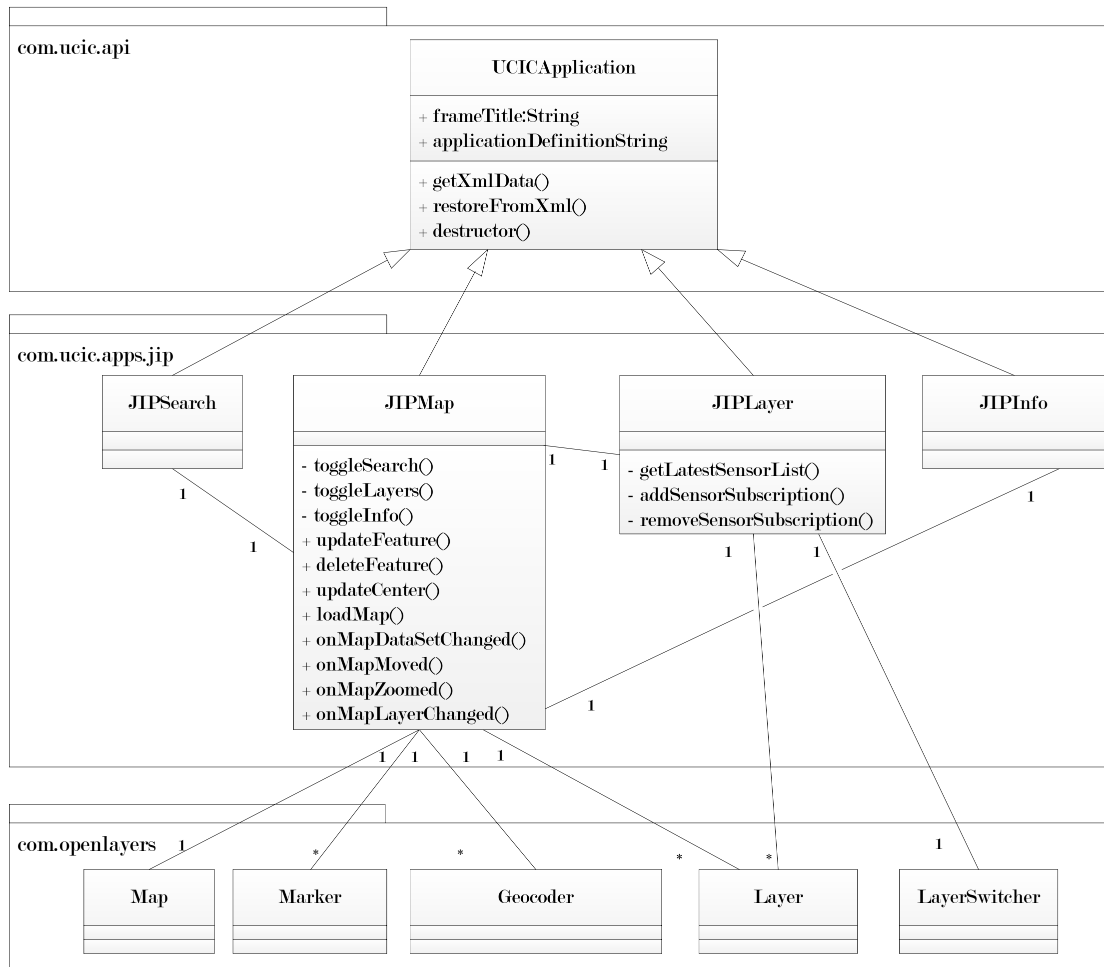

FIGURE 4.8: JIP Application Design.

AJAX API was first considered for this role, and while it offers many benefits (including that the service and developer tools are available at no charge), it also has some shortcomings. Its simplified interface, while pleasant to use, omits support for the addition and selection of user-defined layers, such as layers based on the Web Map Service (WMS) standard defined by the OGC. The API available to developers requires obtaining a key from Google, which is only functional on one domain. Another limitation of the Google Maps API is that the API code itself cannot be deployed on servers not owned by Google, meaning that developers have to rely on Google's uptime and availability. This also makes it impossible for developers to demo their browser-based map application in an area without Internet access. The open source OpenLayers API [79] was therefore selected as the mapping API for the UC-IC JIP application as it does not have these restrictions. From a usability point-of-view, OpenLayers is very similar to Google Maps and, like

Google Maps, it uses AJAX to dynamically load sections of the map as the user drags the map with the mouse cursor.

The *JIP Map Window*, implemented as the *JIPMap* class, is the main window that appears when JIP is selected from the UC-IC applications list. It contains buttons for hiding and showing the other three windows that make up JIP (via the functions *toggleSearch*, *toogleLayers*, and *toggleInfo*). Inviting other users to collaborate on a *JIP Map Window* allows all of those users to see actions such as zooming (synchronized via *onMapZoomed*), panning (*onMapMoved*), changes to sensor sources (*onMapDataSetChanged*), and changes to layers (*onMapLayerChanged*). In addition, users are collaboratively able to draw on the map via annotation tools provided for all UC-IC applications by the UC-IC platform. Users can take comfort in knowing that UC-IC ensures that all users see the same map view at all times.

Other JIP Windows can also affect the contents of the Map Window; for example, using the *JIP Search Window* can invoke the *JIP Map Window* to show a specific location by calling the *loadMap* function. As shown in Figure 4.9, this function obtains the latitude and longitude values of a given string by using the *Geocoder* object of the OpenLayers API and passing the obtained values to the *updateCenter* function of *JIPMap*. In addition, markers based on real-time sensor data are displayed on the map by using the *Marker* object of the OpenLayers API. The markers can move to reveal updated latitudes/longitudes of a GIS feature, or be clicked on to display other detailed visualizations in the *JIP Info Window*. The *updateFeature* function is called by the server side to display real-time markers on the map once the user has subscribed to a sensor (subscribing is done via the *JIP Layers Window*). Both functions take parameters containing the feature's name, latitude/longitude, sensor data, and an image URL for the marker icon to be displayed on the map. Updates in the latitude/longitude values will animate the marker on the map, while updates in the sensor data will affect the contents of the *JIP Info Window* when the marker is clicked. In addition, a *deleteFeature* function is used when the user has unsubscribed from a sensor.

JIP Layers Window By using the *LayerSwitcher* and *Layer* classes of the OpenLayers API, the *JIPLayer* class allows users to choose from a list of map base layers. By default, the list contains several custom map layers conforming to

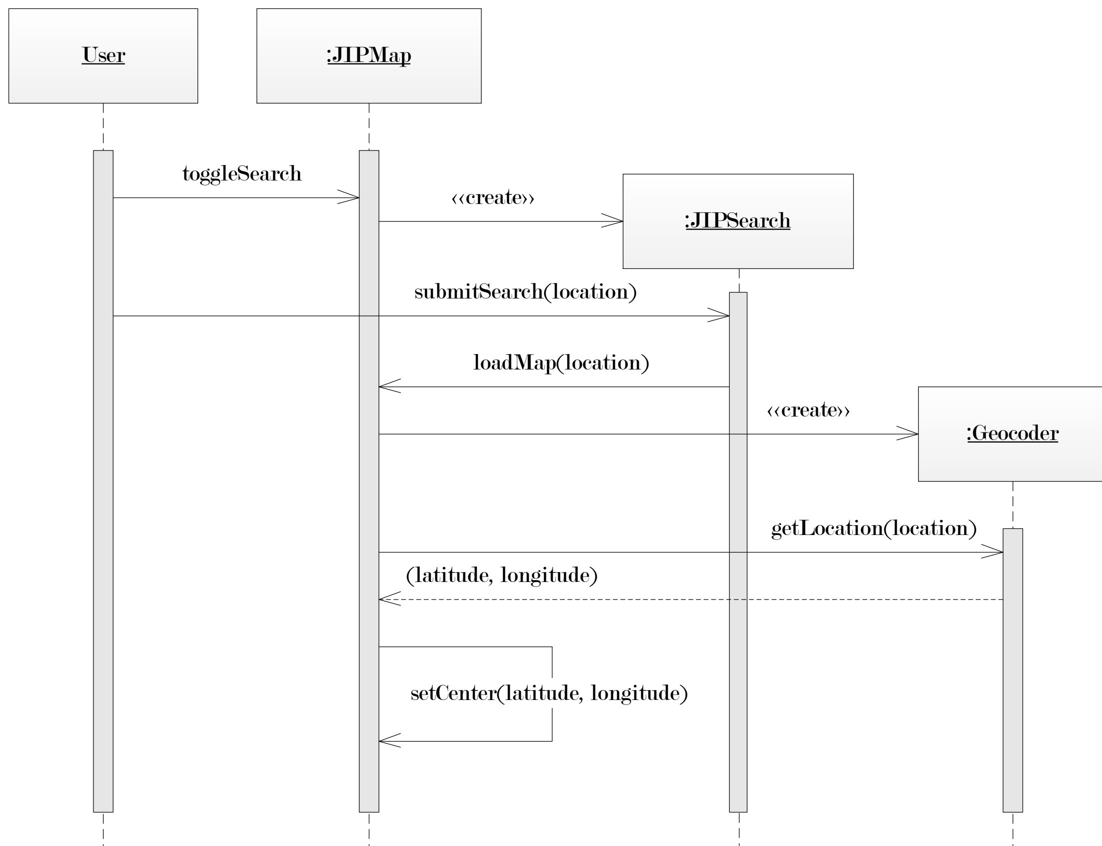

FIGURE 4.9: JIP Geocoding Sequence.

the OGC WMS standard and hosted on a local GeoServer [80] deployment, as well as free WMS layers from NASA and OpenSteetMaps [81][82]. In addition, the OpenLayers API supports loading layers from commercial services such as Google Maps. The *JIP Layers Window* also contains checkboxes for showing and hiding the real-time sensor data markers that are to appear on the map. The list of sensors is retrieved via an HTTP request to the Sensor Registry service of the architecture from the *getLatestSensorList* function. By selecting a sensor, the *addSensorSubscription* function is called, which sends an AJAX request to the server-side *SensorSubscriber* to subscribe to the specific sensor and begin receiving real-time data from it (in the form of calls to *updateFeature* in JIPMap).

JIP Info Window The *JIP Info Window* contains sensor information and data based on user clicks in the Map Window. For example, a user can click on a barometric sensor marker on the map and see a real-time dial or time graph indicating barometric pressure in the *JIP Info Window*. Like all windows in the UC-IC environment, the *JIP Info Window* can be shared with other

users on its own (for example, if a receiving user only needs to watch the live dial move and does not need the corresponding map). The *JIP Info Window* is implemented in the *JIPInfo* class.

By using the *JIP Search Window*, *JIP Map Window*, *JIP Layers Window*, and *JIP Info Window*, users can access real-time sensor data from within the UC-IC web-based environment. UC-IC allows multiple instances of each application to exist, and each application window can be shared with multiple online users, thereby making powerful collaboration scenarios possible that can make use of a variety of real-time sensor data. UC-IC is therefore an ideal environment for advanced GIS collaboration since it can ensure the correct users have all the real-time information they require to make important decisions.

4.2 A Real-Time Platform for Collaborative Multimedia

A real-time platform for collaborative multimedia called Watch Together has been developed that is in many ways similar to UC-IC. Much like UC-IC, the goal of Watch Together is to achieve real-time collaboration between users who, as a group, share a collaborative session. This implies that all users in the session will have the exact same state of the system. For example, all users within the same session will see the same video (at the same playback time), the same image, or the same page in a document or slide. Actions performed by a user (such as changing the image, fast forwarding in the video, or changing the page) are replicated across all the users in order to ensure that the same state is maintained.

Instead of providing an entire collaborative online desktop like UC-IC, Watch Together was created as a lighter version that allows for quicker development of synchronized applications. The client side is developed using Adobe Flash, and Watch Together provides an API which applications use to synchronize data among the users of an established session (only one collaborative session is permitted per user at a time). Flash also allows Watch Together to provide video/audio chat via live webcam streams so that users can see and hear each other as they collaborate over the online content. The multimedia capabilities of Flash also mean that maps can contain more interactive and animated content. To make real-time

collaboration possible, the Watch Together client communicates with the Java-based Red5 Media Server (the Application Server in this case) through Adobe's proprietary Real-Time Messaging Protocol (RTMP) [18].

While Watch Together is a collaborative platform on top of which new applications can be developed, Watch Together itself is designed to be embedded within other environments and portals that don't yet have real-time collaboration features. By accessing the user information from an external environment, Watch Together can introduce synchronized collaboration to the existing users of those environments. For example, Watch Together was deployed as a *Facebook Application* to allow Facebook users to collaboratively watch YouTube videos with friends [83].

Watch Together was developed in collaboration with Bogdan Solomon, Bogdan Ionescu and Dr. Dan Ionescu of the University of Ottawa's NCCT Lab. It is described further in [25].

4.2.1 Application Server

The open source and Java-based Red5 Media Server was selected as the RTMP-compliant application server to power the Watch Together Server [84]. The benefits of Adobe's official Flash Media Server over the open source implementation are not significant in the context of the work presented here.

Watch Together can be deployed within other environments and use the APIs of those environments to authenticate users. For internal development, a separate JEE server was set up for authenticating and authorizing the users who wish to use the system. If the system uses an external authenticating/authorization system, such as when it is embedded as a Facebook application, then the JEE server would not be used. However, if no JEE server is used, the JEE server would be replaced by a web server (such as the Apache HTTP Server [85]) whose sole responsibility is to provide the HTML page and the embedded ShockWave Flash (SWF) files necessary for the client to run the application. Once the client is authorized and the SWF files are loaded on the client side, the client connects via RTMP to the Red5 Media Server. After a connection to the Media Server is established, the client can start collaborating with other users.

Figure 4.10 shows how the server side is structured into three main classes: a class which handles client connections and messages from clients (*ServerModule*),

a class which represents a session and is responsible for holding session state and broadcasting session messages to clients (*ServerSession*), and finally a class which represents the clients themselves (*Client*).

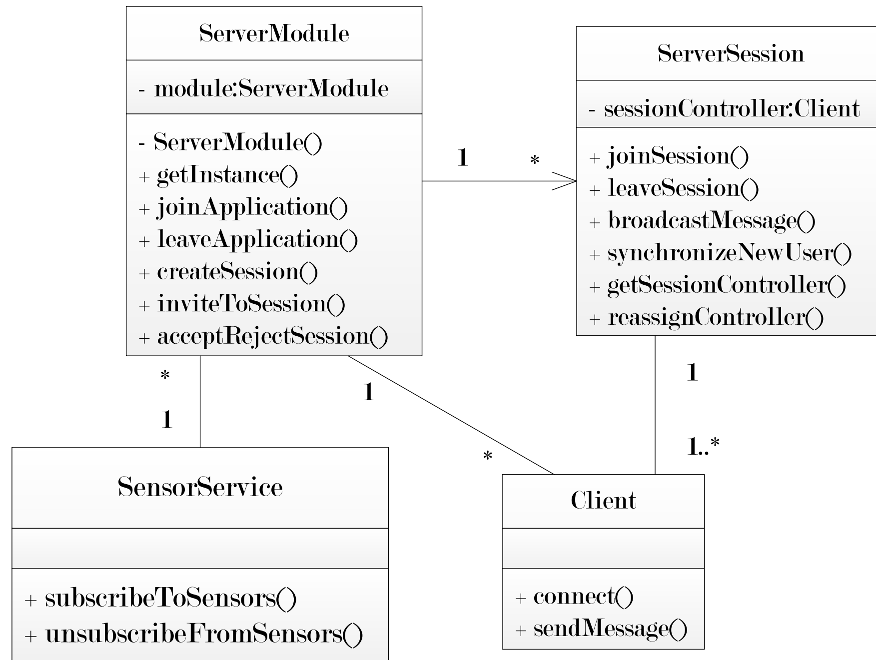

FIGURE 4.10: Watch Together Server Class Diagram.

The *ServerModule* class, which is a singleton for each server, allows clients to join and leave the application. It also allows clients to create sessions and invite other users to sessions. Finally, it allows invitees to accept or reject invitations. The server has zero or more clients connected to it at any time, and has zero or more sessions running at any time. Clients can connect to the server and send messages to other clients. Clients can also join sessions by either creating a new session or accepting an invitation from another client. Clients can also leave sessions. They can do so explicitly or by disconnecting from the server while in a session. When a new client joins a session, the new client is synchronized to the state of the session. Clients can also use a session to broadcast messages to other clients. Finally, in order for users to have some level of control over privileges in the session, every session is given a *sessionController*. The *sessionController* can allow or deny other users from controlling what is seen in a session. Initially, the *sessionController* is the user who creates the session. The *sessionController* can also reassign the session control either explicitly by selecting another user to pass control to, or by leaving the session. If the *sessionController* leaves the session, then another user in the session is randomly chosen by the server as a

sessionController, and all the users in the session are notified. The Red5 Media Server also contains a *SensorService* class that accepts an XML file over RTMP containing a subscribe or unsubscribe request and the associated sensor locations, identifiers and observed properties that are of interest to the client. The class then subscribes to the appropriate sensors through the Publisher/Subscriber Server and notifies the Smart Publisher component of the subscription changes. This is discussed in greater detail in Section 4.3.

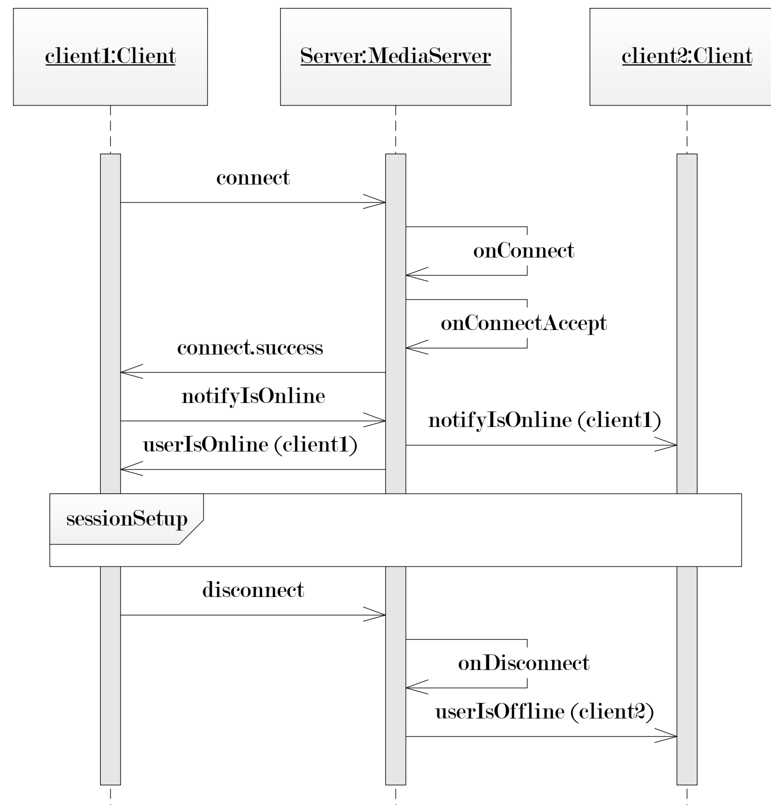

Figure 4.11: Watch Together Connection Message Sequence Chart.

Due to the modular and extendable nature of the client, the communication between the client and the server must be able to support messages that were not considered at design time. Figure 4.11 shows the sequence diagram for establishing the connection between the client and the server, and then disconnecting. The diagram assumes that there is already a client connected to the server (namely, *client2*). The sequence is initiated when a different client, *client1*, connects to the server. Upon connection, the Media Server performs access control and determines if the user should be allowed to connect. If the user is allowed to connect, a *connect.success* message is sent back to the user's client (*client1*). The client then sends a *notifyIsOnline* message, which contains a list of the unique IDs of the client's contacts. Assuming *client2* is a contact of *client1*, the server posts

messages to both *client1* and *client2* that the other client is online. Following a collaborative session, *client1* disconnects from the Media Server. The server then notifies *client2* that *client1* has gone offline.

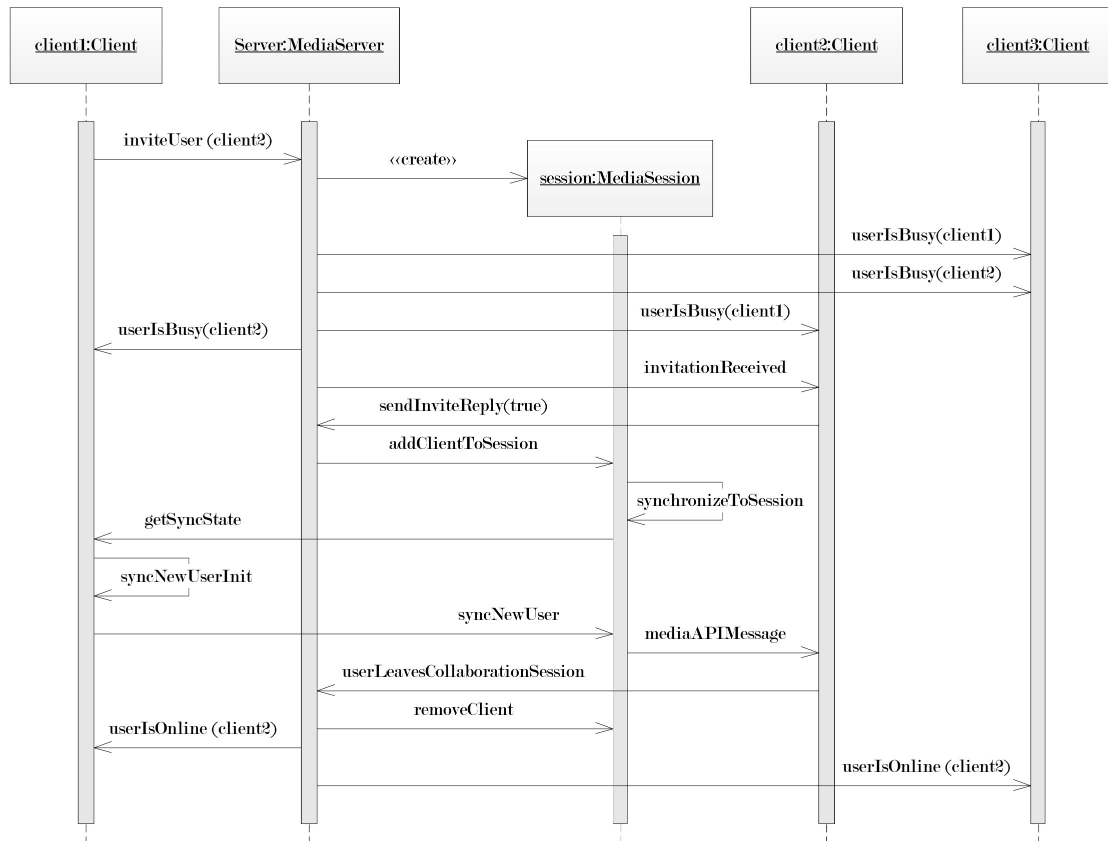

FIGURE 4.12: Watch Together Invitation Message Sequence Chart.

The session sequence diagram for a collaborative session in Watch Together is shown in Figure 4.12. In this example, it is assumed that *client1*, *client2* and *client3* are all contacts of each other. In order to begin a session, a client (*client1* in this case) invites another client, *client2*, to a collaborative session. Upon receiving the request, the Media Server creates a new session, adds *client1* as the controller of the session, and forwards the invitation to *client2*. At the same time, it notifies *client3* that the other two clients are busy, as well as notifies *client1* that *client2* is busy, as well as *client2* that *client1* is busy. The goal of the busy state for clients is to prevent a client from receiving invitations while in a session or while an invitation is pending. After the invitation is received, *client2* decides to accept the invitation and the invite reply message is sent to the server. The server adds the new client, *client2*, to the session. When a new client is added to a collaborative session, the session synchronizes the new client to the state of the session. In order to determine the state of the session, the session asks

the *sessionController* (*client1*) for its state. The *sessionController* determines its state and sends it back to the server-side session (in this case via *syncNewUser*). The server session then sends a *mediaAPIMessage* to the client or clients that are to be synchronized. These are the same messages that are used in order to synchronize new actions performed by the users within a session. Also note that if multiple users join a session while the session is waiting for a synchronize state reply from the *sessionController*, the clients are added to a waiting queue and multiple messages are not sent to the *sessionController*. Once the reply arrives from the *sessionController*, the message is broadcast to all the clients waiting for it. This is done in order to minimize the messaging between server and clients as much as possible. Finally, when a client (*client2* in this example) decides to leave a session, *userLeavesCollaborationSession* is called on the server. When the server receives this message, *client2* is removed from the session and the other clients receive notifications that *client2* is "online", indicating that the client has become available for invitation once again.

4.2.2 Collaborative Web Client Platform

The Watch Together Collaborative Web Client Platform allows developers to create new collaborative applications oriented around online data and media sources, such as YouTube videos and Flickr photos. The Watch Together user interface is composed of three key sections that are always visible:

- A *Content Section* contains the application content that is synchronized among all users in a collaborative session. It displays the Media Modules, which contain the controls and search functionality necessary for interacting with the content. This section takes up the majority of the Watch Together application's user interface.
- An *Icon Section* contains icons representing the different applications available for collaboration. It also provides access to a *User List Module* for seeing which other users are online and inviting them to a session. Selecting any icon from this section will therefore modify the Top Section to display the appropriate content.
- A *Session Section* contains boxes that indicate the users in the current session and displays their live webcam streams. It also provides methods for a

user to enable and disable their own webcam, as well as indicates who the current *sessionController* is and if they have applied any usage restrictions to others in the session.

The Watch Together Platform was developed using Adobe Flex [17]. It features a modular design in order to ensure that the client side is easily extendable and capable of loading its components on demand. By using Flex Modules, modules are dynamically downloaded at run time as they are required. The client side modules are split into "Domain Modules" and "Media Modules":

Domain Modules Domain Modules are used for dealing with domain-specific data within Watch Together. A "domain" represents a deployment instance of the system and each domain has a separate login approach. For example, the login approach for a client integrated as a Facebook application is different than when the deployment uses a local JEE server. There are two Domain Modules which are developed for each domain: *Login Module* A Login Module performs the login logic and retrieves the local user's information and contacts. *User List Module* A User List Module visually displays the user's contacts within the Watch Together interface. The reason for using different *User List Modules* is that different domains can have different contact categorizations. For example, Facebook contacts are called "friends"; a user has a number of friends and there are no subcategories. If the system is deployed within an organization, however, the organization may define groups of users for various tasks. As such, the contact display list has to be capable of showing the contacts within a group, and to allow the user to select a different group. In order to maintain a consistent Look and Feel while making allowance for such functionality differences, all *User List Modules* extend an existing component.

Media Modules Media Modules are used for developing collaborative applications and dealing with their data sources. Media Modules are displayed within the Content Section of Watch Together. There are four Media Module types: a *Search Module*, a *Viewer Module*, a *Control Module*, and an *Information Module*:

- The *Search Module* allows the user to search for a specific media item to share with other people. An external API of the media/data source is

typically used to retrieve the search results. For example, for a YouTube application, the search component mimics the standard YouTube search options (search videos from "Today", "This Week" etc.) and displays a list of thumbnails for the videos. The user can select the video to view by clicking on it from this list. Before the user searches for anything, the thumbnails for the day's most popular videos are given by default.

- The *Viewer Module* displays the actual media content selected from the list provided by the *Search Module*. For YouTube, this is the actual streaming video. The viewer supports a maximized full-screen mode and is responsible for resizing the media content.
- The *User Control Module* displays the various controls that the user requires to interact with the media content. Actions triggered via this component are typically synchronized with other users, although this is not always the case. For YouTube videos, the *User Control Module* contains play/pause, volume and video timeline seeking controls. However, the volume is not synchronized across the session, as various users might prefer different volume settings.
- The *Information Module* displays useful information related to the currently selected media content. In the case of YouTube, this is simply the title of the video.

Each of the four Media Module types implements the Model-View-Controller (MVC) software pattern. The *model* is shared between all four components as it represents the shared state of the media content.

Figure 4.13 shows the activity diagram of the client. Initially, the client's browser loads the SWF file from the server. Once the main application is loaded, it determines the domain under which it is deployed. Based on the domain, it loads a configuration file from the server. This XML configuration file describes the Domain Modules to load, as well as the Media Modules which are available to the application. The client code then loads the *Login Module* and the *User List Module*, and performs a login for the user. As part of the login process, the user's contacts are loaded. Once the user is logged in and the contact list is loaded, the client opens the connection to the Media Server and the *Search Modules* become available. When the user selects a *Search Module* from the list of available modules, the appropriate *Search Module* is loaded from the server. Finally, if the user

selects a specific media item to view, the corresponding *Viewer Module* and *User Control Module* is loaded from the server.

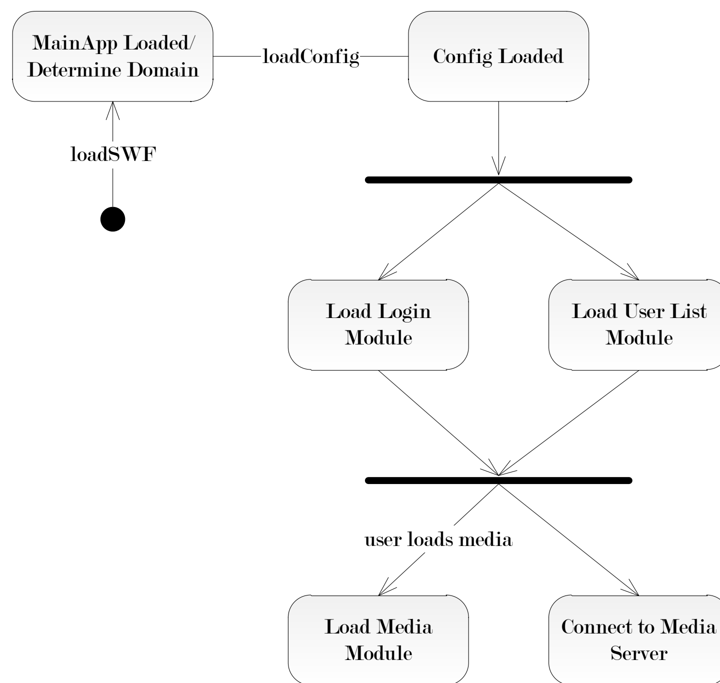

FIGURE 4.13: Watch Together Client Activity Diagram.

As was mentioned, each Media Module implements the MVC software pattern. In order to allow for the extendability of the platform, each of the *controllers* for the modules must implement one of the provided interfaces. This is done in order to ensure that the modules, which are loaded at runtime, can communicate with each other. The *model*, which is common between all four modules, is defined for each media type by the developer as it is media-dependent. The *viewer* for each of the modules is also media-dependent, and as such is defined by the developer. Figure 4.14 shows the structure of the client code interfaces.

The *SearchController* performs two actions: *search*, which uses the underlying search mechanism for the data source in order to find media content and is dependent on the data source, and *loadMedia*, which loads the selected media content into the *Viewer Module*. In order to actually load media, a *MediaCommandQueue* is used, which is a singleton. The role of the *MediaCommandQueue* is to store commands until the *Viewer Module* is loaded and then play the commands in the order received. All commands, be they *loadMedia* or control commands, pass through the *MediaCommandQueue*. This is necessary due to the fact that modules are loaded on demand. If a new user joins a session, receives a synchronization

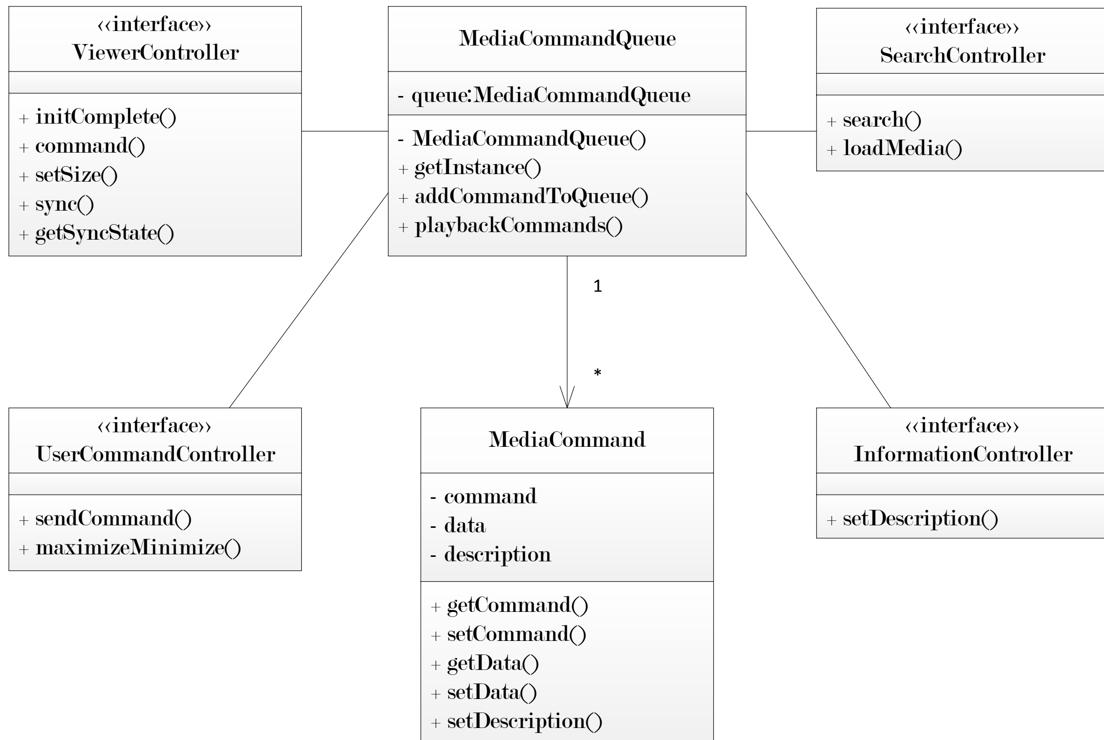

FIGURE 4.14: Watch Together Client Interfaces.

command, and new commands are received while the user is still loading the correct *Viewer Module*, the newer commands would be lost, which would lead to the desynchronization of the sessions. A second role for the *MediaCommandQueue* is that of ensuring that the correct modules are loaded. When a new command is received, the *MediaCommandQueue* determines if the currently loaded modules can perform the command. If they can not, then the correct modules are loaded, the queue is emptied (since commands affecting the loaded modules no longer apply while the new modules are loaded) and the command is added to the queue. If, while modules are being loaded, a new command comes which requires different modules than the ones being loaded, then similarly the queue is emptied and the command is added to the queue. The *MediaCommandQueue* is also responsible for broadcasting messages to other members of the session through the connection to the server. To store a command, the *MediaCommandQueue* uses a *MediaCommand* data structure which has three fields: *command*, which represents a unique ID for the command (for example loadVideo, play, pause, seek for video media content); *data*, which stores the data for the command (for example the new time position for the video seek command); and *description*, which holds the description of the media type used by the *Information Module*. The *ViewerController* performs five actions: *initComplete*, which notifies the *MediaCommandQueue* that the

Viewer Module has finished initializing and that the command queue should start playing back commands; *command*, which performs a command coming from outside (either from another user or from the *User Control Module*); *setSize*, which resizes the viewer; *sync*, which synchronizes the *Viewer Module* to the current state of the session (this is executed as a new user to a session); and *getSyncState*, which retrieves the current state of the session (this is executed by the user who is the *sessionController* when new users join). The *UserCommandController* performs two actions: *sendCommand*, which sends a command to the *MediaCommandQueue* to be executed by the viewer, and *maximizeMinimize*, which is sent to the *Viewer Module* to resize it. Finally, the *InformationController* has only one method, *setDescription*, which is called from the *MediaCommandQueue* when new media content is loaded.

Through the use of these "MediaAPI" interfaces, developers can easily add new applications and data sources to the Watch Together Collabroative Web Client Platform without needing to worry about the synchronization mechanism. The modular approach also allows different Media Modules to be distributed across different locations, thus behaving like a cloud application. Developers can host their own modules either on their own servers or on clouds like Amazon's Elastic Computing Cloud (EC2), and the system, through a simple configuration file, can find and load them. This can be easily extended to use a registry instead of a configuration file in order to discover and load modules. Currently, Watch Together has applications supporting YouTube recorded videos, Flickr and Facebook images, Twitter text messages, local documents where users can upload documents to the system and share them, and UStream live video.

4.2.3 GIS Web Application

Adding a new application to Watch Together requires using the provided interfaces of the MediaAPI and integrating with an external data source. For the GIS Web Application, the Google Maps Flash API was selected for its extensive Flash-based support and instant familiarity to most users. A Google Maps application was therefore created for Watch Together which is capable of displaying real-time sensor data for collaboration among all users within the same session.

The GIS Web Application design shown in Figure 4.15 was implemented by overriding the methods defined in the interfaces introduced in Figure 4.14. Each of the Media Modules was therefore implemented as follows:

- The *Search Module* of the Google Maps Watch Together application contains a text box and search button for specifying the location that is to be displayed within the *Viewer Module*. The *SearchController* interface is implemented by a *GoogleMapsSearch* class to provide this functionality.

- The *Viewer Module* was created by implementing the *ViewerController* interface in a class named *GoogleMapsViewer*. The *GoogleMapsViewer* class creates the main *Map* object via the Google Maps API. Event listeners are configured to create synchronized calls when the center of the map, zoom level of the map, sensor dataset, or base layer are changed (via *onMapMoved*, *onMapZoomed*, *onMapDataSetChanged*, and *onMapTypeChanged*, respectively). Google's standard Map, Satellite and Hybrid base layers are provided as possible base map types. Sensors are displayed as animated markers via the *Marker* class of the Google Maps API. These markers can be clicked on to reveal bubbles containing the real-time data of the sensor. *GoogleMapsSearch* calls the *loadMap* method of *GoogleMapsViewer* with its search string, which *GoogleMapsViewer* translates into latitude and longitude values via the *ClientGeocoder* class of the Google Maps API before centering the map on those values.

- In the context of the GIS Web Application, the *User Control Module* is responsible for providing a list of sensors that the user is able to have appear on the map. By marking a checkbox near a sensor's information in a provided list, a user is able to subscribe to the sensor's live data stream. The list of sensors is obtained via a service of the architecture's Sensor Registry. Subscribing to a sensor is performed by sending a RTMP message containing the sensor's ID to the Red5 Media Server. The subscriber component of the server side then subscribes to the data stream from the Sensor Server (if it's not already subscribed because of another client). The server side then calls the *mapSensorDataResult* method of the *GoogleMapsViewer* class to pass the real-time sensor data as it becomes available (in fact, the method is private since it receives the data through an event from the Watch Together class handling all server-side communication). The *GoogleMapsViewer* can then

create a marker and display the sensor's real-time data to the user. The *User Control Module* also contains buttons for minimizing and maximizing the map. This is done within a *GoogleMapsUserControl* class, which implements the *UserCommandController* interface. The main map controls (zooming, etc.) are instead provided by the *GoogleMapsViewer* in an interface that is familiar to users.

- Finally, the *Information Module* simply displays the name of the selected marker. The *InformationController* interface was implemented in a *GoogleMaps-DisplayInfo* class to make this possible.

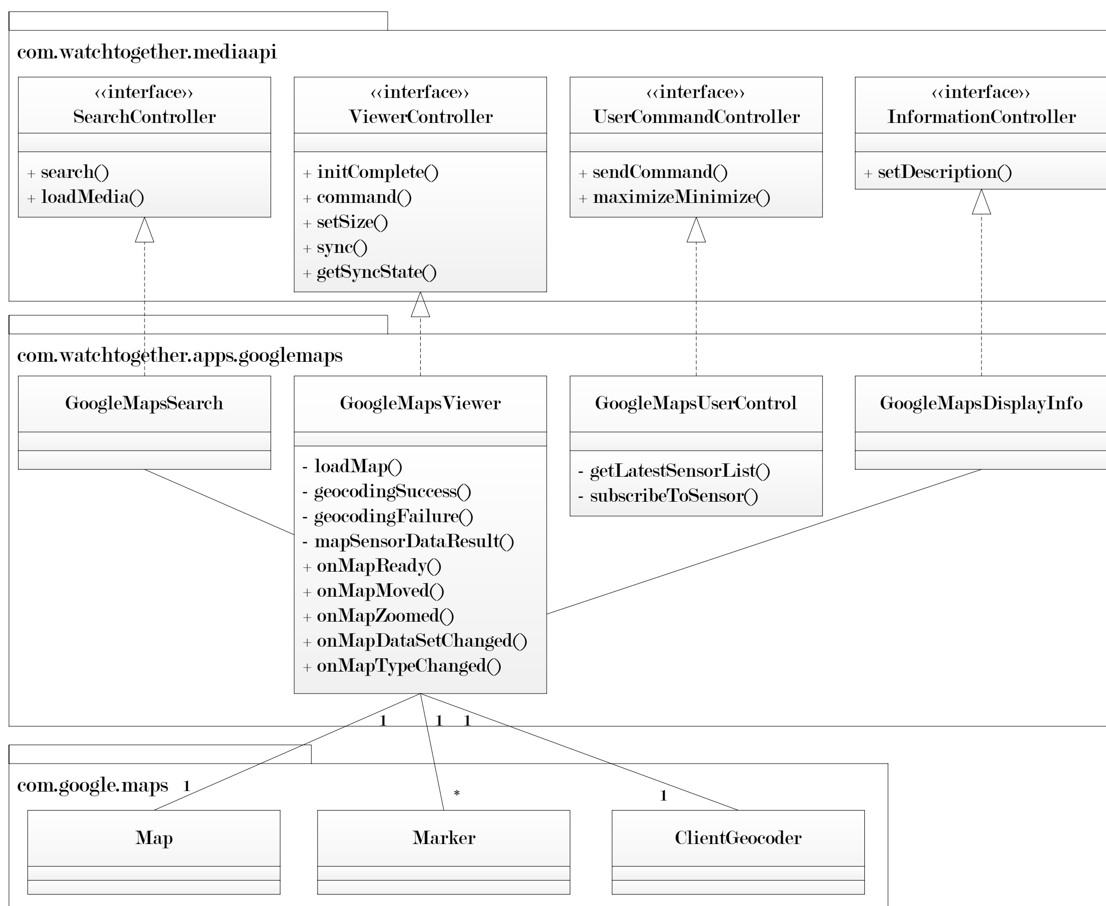

Figure 4.15: Google Maps Application Design.

By taking advantage of the Watch Together platform's flexibility in integrating external APIs like Google Maps, the GIS Web Application built with the MediaAPI offers an easy-to-use collaborative map browsing experience. Since the Flash platform excels at bringing interactive and animated content to the web, live sensor data appears as smooth animations and users can use webcam chat to discuss

their GIS findings. Watch Together's Google Maps application therefore makes real-time sensor data collaborative and user-friendly.

4.3 Shared Components

This section contains the implementation details for the components of the architecture that were very similar for both the UC-IC and Watch Together implementations. These components use open standards to ensure real-time data is delivered to the client side in a timely and reliable manner.

4.3.1 Subscriber

The Subscriber component of the architecture was implemented as shown in Figure 4.16. The Java class named *SensorService* was integrated within the Application Server of the UC-IC and Watch Together environments as previously shown in Section 4.1.1 and Section 4.2.1, respectively. The *SensorService* class receives subscription requests from clients and creates a Subscriber by using the *createSensorListener* method of the *SensorDataSubscriber* class. The *SensorDataSubscriber* class follows the Java Message Service (JMS) standard for connecting to a JMS Server as a Subscriber. This includes obtaining a Java Naming and Directory Interface (JNDI) connection to the JMS Server. JNDI allows a variety of naming services to be accessed using a standard Java API.

To connect to the JMS Server, an *InitialContext* object is created by specifying its host and port information via a "java.naming.provider.url" property. This connection to the JMS Server is a JNDI connection. This JNDI connection is used so that the Subscriber can look up a *ConnectionFactory* object in order to create a *Session* object. The *Session* object is used to create an object which implements the *MessageConsumer* interface for the specific *topic* that the Subscriber wishes to subscribe to (and that the Publisher will be publishing to). Once a *ConnectionFactory* is obtained, a *Connection* object to the JMS Server can be created by specifying a username and password for the JMS Server.

This *Connection* can then be used to receive messages in real-time as they become available to the JMS Server via the Publishers. In addition, a *SensorDataMessageListener* object is created (which implements the JMS *MessageListener* interface)

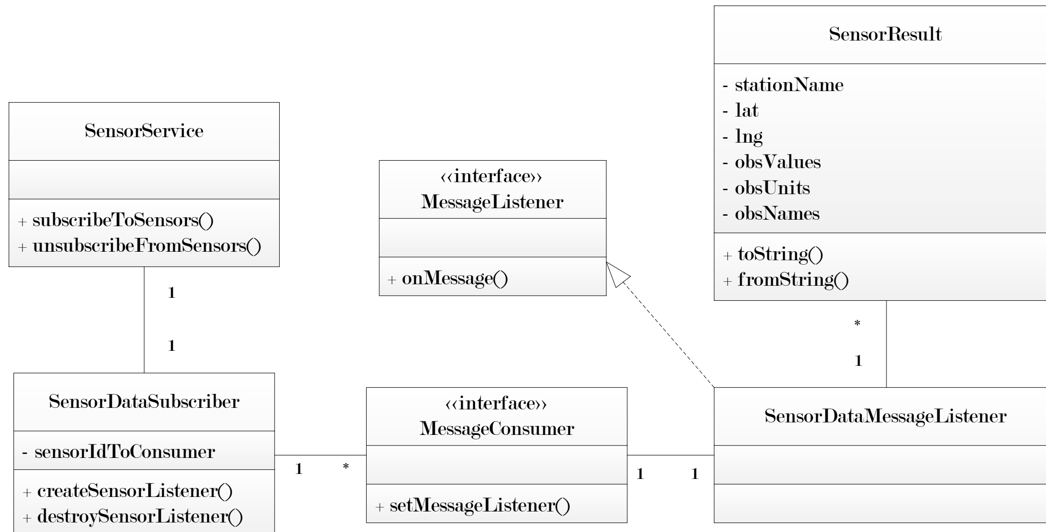

FIGURE 4.16: Subscriber Class Diagram.

such that the *onMessage* event is processed by the *SensorDataMessageListener* when the *MessageConsumer* receives a message from the JMS Server. The JMS message (which is actually a string) is then deserialized into one or more *SensorResult* objects and forwarded to the appropriate method on the client (using AJAXPush for UC-IC to call *updateFeature* of *JIPMap* and RTMP for Watch Together to call *mapSensorDataResult* of the *GoogleMapsViewer*). This sequence of events is summarized in Figure 4.17.

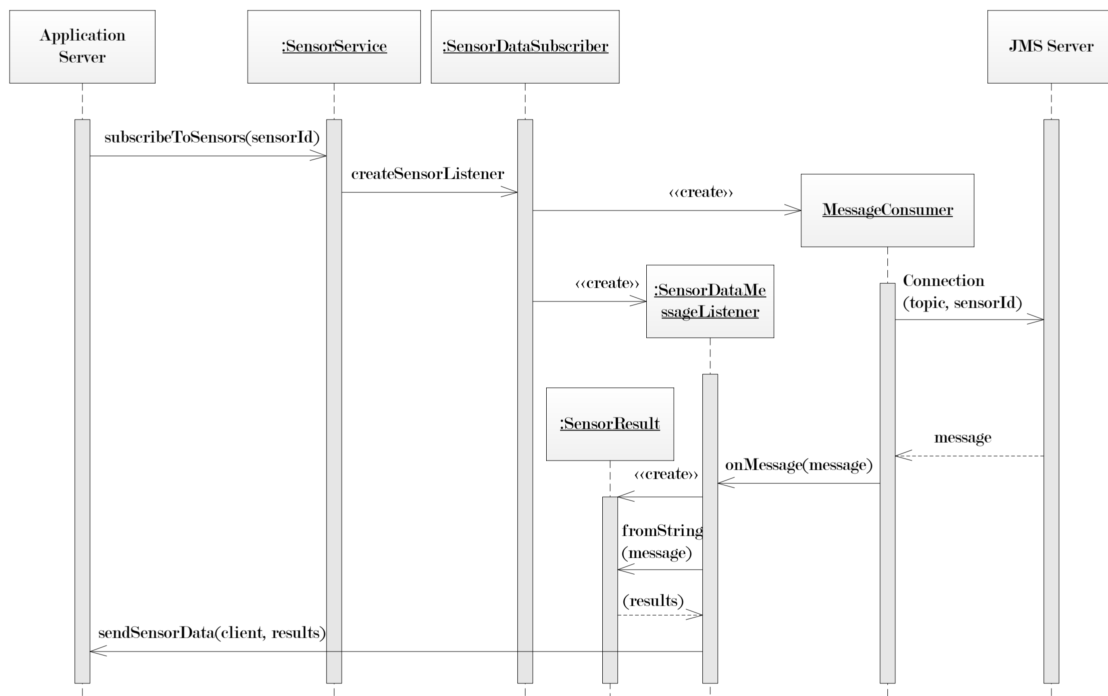

FIGURE 4.17: Subscriber Message Sequence Chart.

When the user selects a sensor from the client-side list, an XML-based message using a custom format is passed to the *subscribeToSensors* method from the *SensorService* class through the Application Server. This format will be called "Custom XML" and an example of it is shown in Listing 4.1. The <sensors> tag of the message specifies if the request is of type "subscribe" or "unsubscribe". The <sensor> tag within the <sensors> tag specifies the unique *sensorId* of the sensor, as well as the *URL* of the SOS service that is to be subscribed/unsubscribed to. Finally, the <property> tags within the <sensor> tag are used to list the *observed properties* that the user is interested in receiving results for. The client side is able to complete this information based on the data returned from the Sensor Registry component of the architecture.

```
<sensors method="subscribe">
   <sensor id="urn:x-noaa:def:station:NOAA.NOS.CO-OPS::1611400" url="
   http://opendap.co-ops.nos.noaa.gov/ioos-dif-sos-test/SOS">
      <property>AirTemperature</property>
      <property>BarometricPressure</property>
      <property>Currents</property>
   </sensor>
</sensors>
```

Listing 4.1: Custom XML for Subscribing/Unsubscribing.

A "subscribe" request causes the *SensorService* class to create a new *MessageConsumer* instance (via the *createSensorListener* method of *SensorDataSubscriber*) for each unique *sensorId*. This occurs only if the subscription has not already been created by a different client, in which case the same *SensorDataSubscriber* is used. Rather than creating a new topic for every sensor, the JMS *filter* feature (which is specified along with the topic when defining the *MessageConsumer*) is used to ensure each *SensorDataMessageListener* (one per *MessageConsumer*) is only notified of messages pertaining to a specific *sensorId*. The JMS Server selectively sends messages to a *MessageConsumer* based on the filter conditions for this one topic [86].

The *SensorService* class is also responsible for notifying the Smart Publisher directly about the client's interest in subscribing to the sensor data. It does this by sending the same Custom XML it received from the client side (see Listing 4.1) to the Smart Publisher. The Smart Publisher can then begin polling for real-time

data from the Sensor Server by using the server's location and other details from the Custom XML. It will then publish the data via the JMS Server.

A Custom XML message containing "unsubscribe" is sent from the client when it no longer wishes to receive data for a specific sensor. If there are no other users subscribed to the sensor, the *SensorDataSubscriber* is destroyed and the Smart Publisher is notified so that it may stop polling. In addition, if the Application Server detects a client disconnecting, it notifies the SensorService, which creates an "unsubscribe" request for the Smart Publisher. This way, the Smart Publisher is only publishing sensor data if there are clients to receive it.

4.3.2 Publisher/Subscriber Server

The Publisher/Subscriber Server component is responsible for routing messages containing sensor data from the Smart Publishers to the appropriate Subscribers. Due to their Java-based Application Servers, both UC-IC and Watch Together are able to have JMS Subscribers that connect to a server running the JMS implementation from JBoss Messaging called HornetQ [87]. HornetQ is an open source messaging system that supports important features such as clustering and has undergone extensive tuning so that it can provide very high asynchronous messaging performance. In fact, HornetQ recently set a new record for JMS messaging system performance using the SPECjms2007 benchmark [88].

The setup of HornetQ within JBoss is fairly simple and includes defining the topic in a configuration file. The topic name used is *sosDataTopic*. Rather than have multiple topics, the *sosDataTopic* is used along with the filter feature of JMS in order to allow multiple subscribers and multiple publishers to exchange a variety of real-time data. The Subscriber uses the *MessageConsumer* interface of the JMS API to register its interest for a specific *sensorId* on this topic. The Smart Publisher, through the (*MessageProducer* interface, transmits messages to the JMS Server on the *sosDataTopic* as they become available from the Sensor Server. The messages contain the real-time data packaged as a string, along with the *sensorId*. Since the Subscribers filter on the *sensorId*, the JMS Server transmits the message only to the *MessageConsumer*-based instances that match the specific *sensorId*. In this implementation, there will only be one subscription per *sensorId* and, in the case where there are multiple clients interested in one *sensorId*, the Application Server ensures that all clients receive its data by managing a list of

clients mapped to a specific *sensorId* via the *sensorIdToConsumer* attribute of the *SensorDataSubscriber* class (shown in Figure 4.16). Figure 4.18 summarizes how three real-time messages from the Smart Publisher reach a Subscriber.

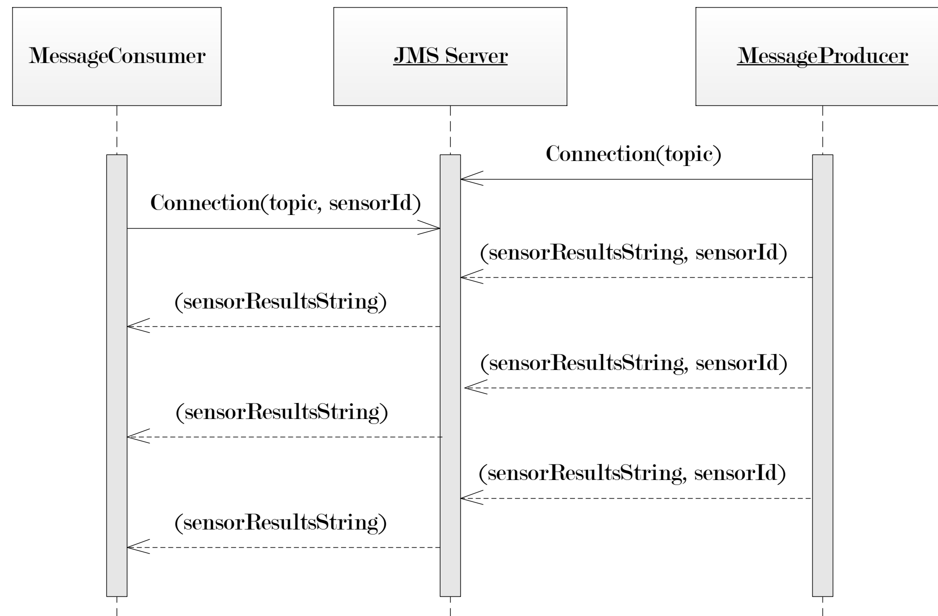

FIGURE 4.18: Publisher/Subscriber Server Message Sequence Chart.

4.3.3 Smart Publisher

The Smart Publisher was implemented as the series of classes shown in Figure 4.19. The classes were packaged and deployed as an independent application within JBoss (unlike the Subscriber which is deployed alongside the Application Server).

The *PublisherServlet* is a servlet that uses the Quartz scheduling service from JBoss to generate “jobs” for retrieving sensor data conforming to the Observations & Measurements (O&M) standard from the Sensor Server. Using such a scheduling service allows for optimal performance when repetitively executing multiple tasks such as polling the Sensor Server. Specifically, the *PublisherServlet* receives the Custom XML (shown previously in Listing 4.1) from the Application Server via its *doPost* method. If the Custom XML message contains a “subscribe” request, the *PublisherServlet* schedules a recurring *SensorDataJob* via a *JobManager* object (one job per *sensorId*). If the custom XML message contains an “unsubscribe” request, the job is destroyed.

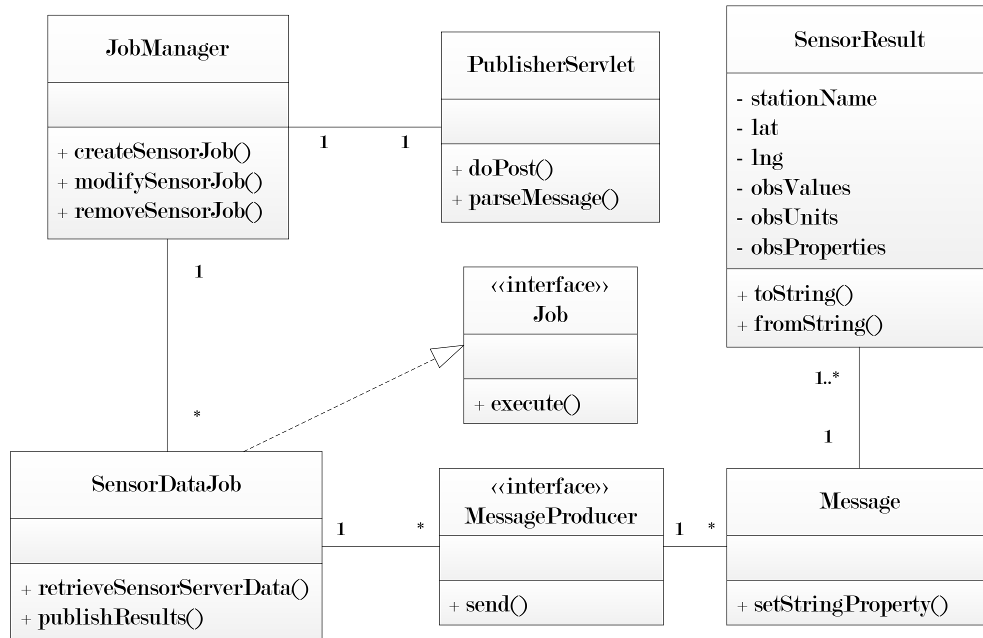

FIGURE 4.19: Smart Publisher Class Diagram.

A *SensorDataJob* consists of executing the *retrieveSensorServerData* and *publishResults* methods. Based on the sensor location, sensorId, and requested sensor properties contained in the XML message, the *retrieveSensorServerData* method formulates a SOS *GetObservation* request to the SOS Server. As shown in Section 2.3.1.2, a response can be obtained from an SOS Server by sending a HTTP GET request to a URL. Assuming the Custom XML from Listing 4.1 is received by the Smart Publisher, the following URL can be assembled inside *retrieveSensorServerData* and sent to the SOS Server:

http://opendap.co-ops.nos.noaa.gov/ioos-dif-sos/SOS?service=SOS&request=GetObservation

&version=1.0.0&version=1.0.0&responseFormat=textxml&observedProperty=AirTemperature

&offering=urn:x-noaa:def:station:NOAA.NOS.CO-OPS::1611400

The sections of the URL that are underlined are completed by using the data received inside the Custom XML, with the unique *sensorId* as the “offering”. The remaining sections are defined as part of the SOS standard [46] and remain constant. A simplified version of the O&M response received is shown in Listing 4.2. This data is then parsed as a *SensorResult* (the value “19.700” is parsed within *obsValues*, “C” into *obsUnits*, etc.) for each *observedProperty* requested. Requests for “BarometricPressure” and “Currents” properties are completed next, as requested via the sample Custom XML within the <property> elements. In

addition, an *observedProperty* may return multiple results (such as WaterTemperature, Heading, Pitch, and Roll for "Currents"), requiring *obsValues* to be an array.

```
<?xml version="1.0" encoding="UTF-8"?>
<om:CompositeObservation xmlns=... >
  <om:procedure>
    <om:Process>
      ...
       <ioos:StationName>Nawiliwili, HI</ioos:StationName>
       <gml:Point gml:id="Station1LatLon">
         <gml:pos>21.9544 -159.3561</gml:pos>
       </gml:Point>
      ...
    </om:Process>
  </om:procedure>
  <om:observedProperty xlink:href="http://ioos.gov/gml/IOOS/0.6.1/
   dictionaries/phenomenaDictionary.xml#AirTemperature" />
  <om:featureOfInterest xlink:href="urn:cgi:feature:CGI:EarthOcean" />
  <om:result>
    ...
   <ioos:Quantity name="AirTemperature" uom="C">19.700</ioos:Quantity>
    ...
  </om:result>
</om:CompositeObservation>
```

Listing 4.2: Sample Response from GetObservation Request.

Once the *retrieveSensorServerData* method of a *SensorDataJob* has completed, the *publishResults* method is executed. As with the Subscriber, publishing is accomplished by establishing a JNDI connection to the JMS Server to look up a *ConnectionFactory* object in order to create a *Session* object. However, the Session object is now used to create a *MessageProducer* for the *sosDataTopic*. Once a *Connection* object is established, a *Message* object can be sent to the JMS Server. The *Message* object contains serialized strings for each *SensorResult* object along with a *sensorId* on which the Subscriber is filtering.

Figure 4.20 shows a message sequence chart for the Smart Publisher where the *PublisherServlet* object is assumed to have received a call to its *doPost* method containing a Custom XML "subscribe" message.

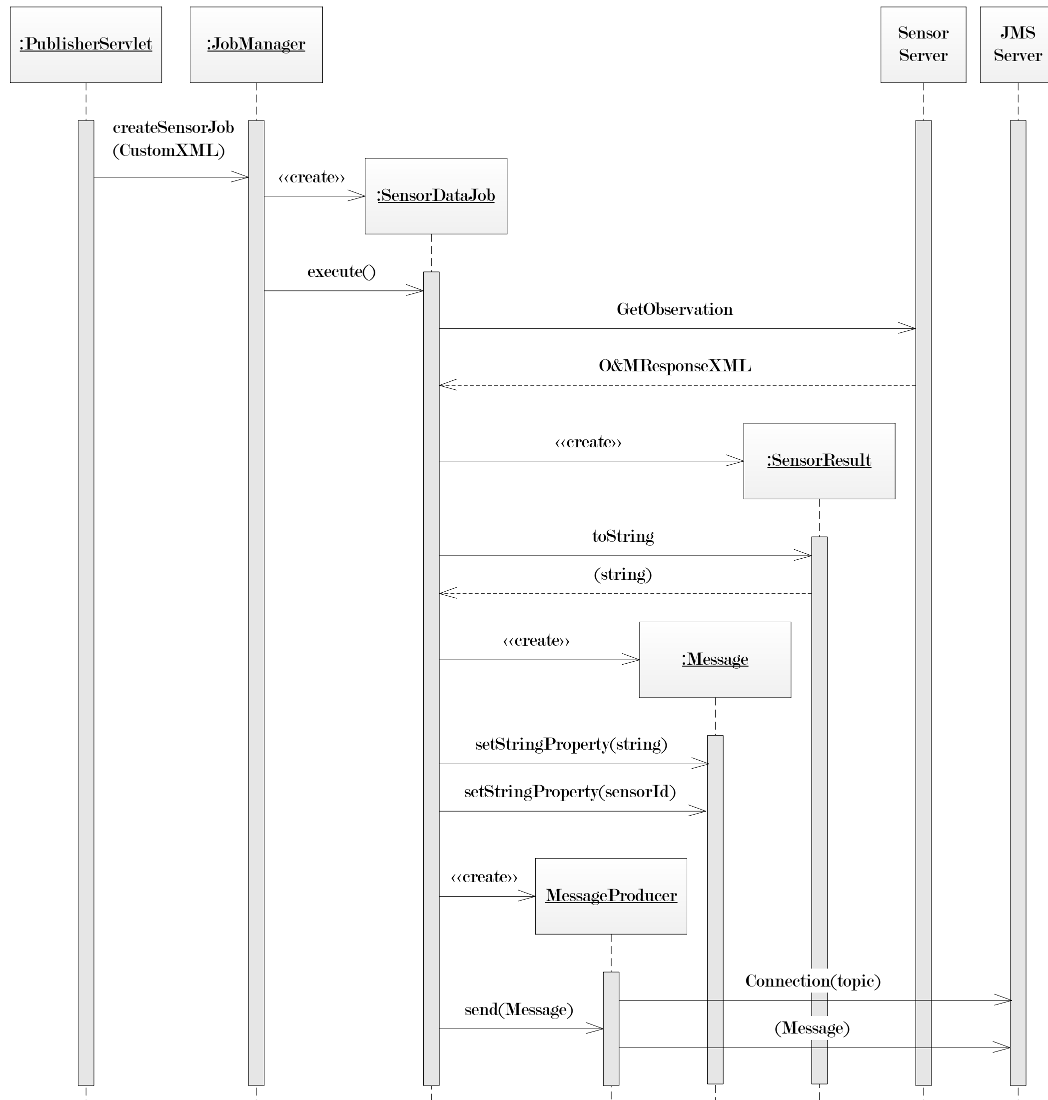

Figure 4.20: Smart Publisher Message Sequence Chart.

4.3.4 Sensor Server & Spatial Database

While the architecture is designed to operate with disparate external sensor sources, a Sensor Server was also set up internally in order to be able to control and observe all the components of the real-time architecture. The Sensor Server component used internally is based on the Sensor Observation Service (SOS) Server project from 52North [52], an open source implementation of the Open Geospatial Consortium's SOS standard. The implementation is Java-based and can be deployed within JBoss.

As defined by the SOS standard, the SOS Server exposes the sensor data as REST-based services, meaning that the sensor data can be accessed through HTTP

requests via a URL. The SOS standard was described more thoroughly in Section 2.3.1.2. The Smart Publisher performs continuous *GetObservation* requests based on the Sensor Server URL, sensor ID, and properties whose results were requested by the client. The SOS Server responds with the resulting values in the O&M format, which are serialized and asynchronously forwarded through the JMS Server to the Application Server, and the Application Server sends them for display on the subscribed UC-IC or Watch Together clients using AJAX or Flash, respectively.

For its Spatial Database, the SOS Server from 52North uses a PostgreSQL database. In addition, the PostGIS extension to PostgreSQL needs to be installed since the database is required to store geographic objects. All data received by the Sensor Server from the Sensor Feeder is stored in the database by default, and the database is accessed to complete all requests from the Smart Publisher. The database is useful for making historic data available to clients by accessing the Sensor Server directly from the client side. Since the primary focus of this thesis is on the real-time transfer of data, such database-driven scenarios were not explored extensively. The database schema and other SOS Server implementation details are available on the 52North website [89].

4.3.5 Sensor Feeder

The Sensor Feeder is, at its core, a very simple component of the architecture. It is responsible for obtaining data from the Sensor and sending it to the Sensor Server such that it can be redistributed and stored. Since a server based on the Sensor Observation Service (SOS) standard is used in this implementation, the SOS standard for submitting sensor data must be followed. The process for using an *InsertObservation* request was shown in Listing 2.1 of Section 2.3.1.2. For example, the <AssignedSensorId> tag is used to specify the unique sensor name, the <om:observedProperty> tag contains the types of properties being observed (such as "temperature", "position", etc.), and the <om:results> tag contains the value observed. Values must be defined using tags that comply with mature standards such as Observations & Measurements (O&M) and Geographic Markup Language (GML), which are thoroughly documented by the OGC [47][48]. The combination of these and other OGC standards make it possible to define and track practically any source of real-time data. An XML response containing an

error message is returned if the InsertObservation request that was submitted to the SOS Server contained syntax errors.

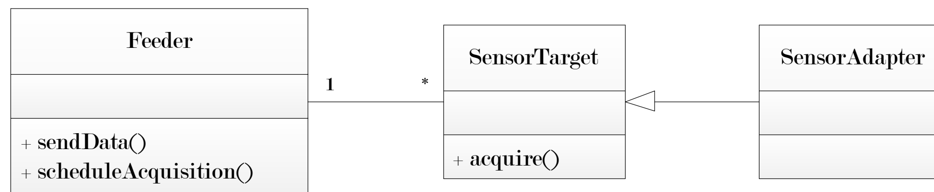

Figure 4.21: Sensor Feeder Class Diagram.

The class diagram in Figure 4.21 shows a simple Sensor Feeder that has been implemented. It consists of a *Feeder* class, which regularly polls a SensorTarget for new updates (the use of the Quartz scheduling API has been discussed in Section 4.3.3). The software Adapter Pattern is used to adapt the data acquisition request to a specific Sensor format or API. For example, the Sensor may be an online real-time data source that needs to be polled using HTTP requests, or a USB Weather Station with a manufacturer-provided API. Whatever the source, a custom *SensorAdapter* needs to be developed to transform the arbitrary data into the well-structured standards-compliant SOS format such that it can be accessed in real-time by web-based clients. The *SensorAdapter* is also responsible for adapting and parsing the response into useable string values. Once the clean sensor data is returned to the *Feeder* class, it is transmitted to the SOS server by using a well-formed *InsertObservation* request. If it is the first acquisition, a *RegisterSensor* request is sent to the SOS server as required by the SOS standard. This basic interaction sequence is shown in Figure 4.22. Properties such as *sensorId* and *observedProperty* can be defined using a separate XML file accessed by the *Feeder* if they are not available via the Sensor directly.

Other SOS-compliant implementations of Sensor Feeders are available, such as one from 52North which uses an extensive XML-based configuration file to define the data source and format, with support for a large variety of known raw sensor data formats and standards. As long as the component is polling a real-time data source and submitting this data to the SOS server as soon as it is received, it qualifies as a Sensor Feeder for this architecture.

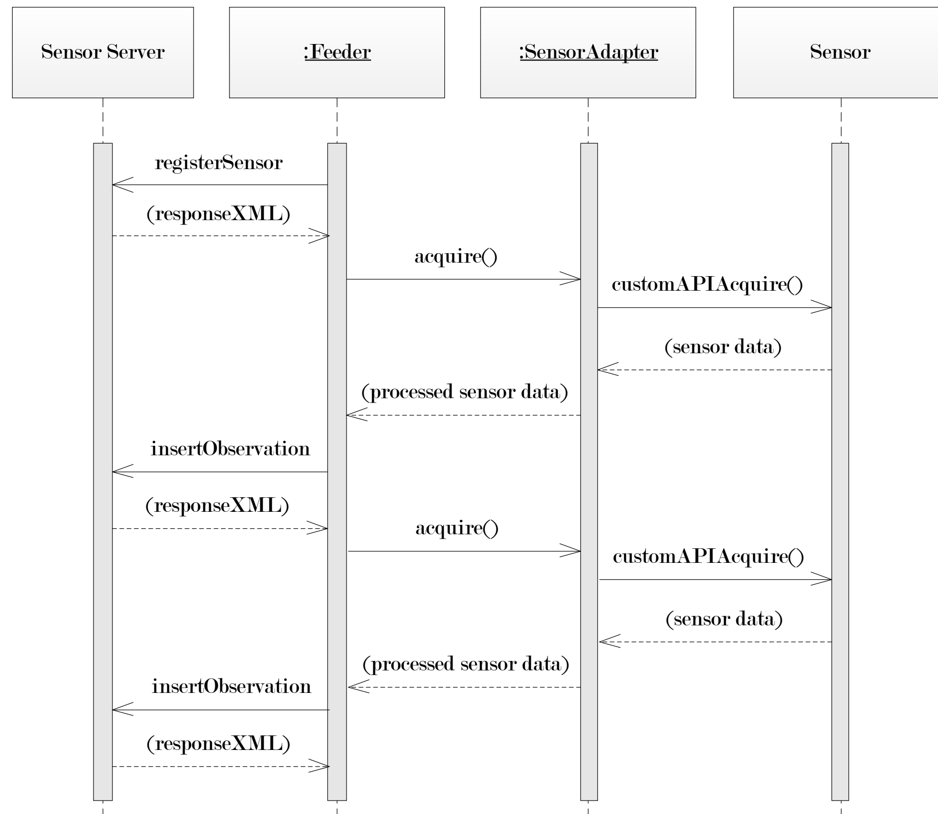

FIGURE 4.22: Sensor Feeder Message Sequence Chart.

4.3.6 Sensor

The Sensor component of the architecture does not itself require an implementation as it is a typically a physical or external source of real-time observation data. It must, however, provide a means to have its real-time data accessed or polled by the *SensorAdapter* of the Sensor Feeder. For a physical transducer, an interface or API is required such that the hardware values can be accessed from Java code. In most cases, however, sensors are deployed by organizations like NOAA [27], who ensure that all their sensors follow a specific sensor standard and make their real-time data available in this standard online. Section 2.3.1 discussed several such standards. While there exist numerous formats and standards used by Sensors all over the world, the flexibility of the SOS standard allows practically all of them to be made available to web clients as long as a *SensorAdapter* can be implemented.

4.3.7 Sensor Network

In this architecture, a Sensor Network consists of a collection of Sensors that can be accessed through a Sensor Server. For example, NOAA provides access to a public Sensor Network consisting of over one hundred real-time sensors that are already served by a SOS Server [27]. To make the data from this Sensor Network accessible from a user's web browser for collaboration with other users, the SOS Server's list of sensors and their properties need to be indexed internally by the Sensor Registry. To discover such public Sensor Networks, a query can be performed on a search engine like Google for the string "inurl:service=SOS" since the standard requires the string "service=SOS" to be present when accessed as a REST-based service. The Sensor Registry can therefore obtain a list of SOS Servers which represent Sensor Networks. By supporting external Sensor Networks in this fashion, users can be presented with large numbers of new real-time Sensors that are available online at no charge. Since the data is also collaborative in the web client, users can share their discoveries with other contacts. This component therefore illustrates the advantages of using open GIS standards in the architecture, as well as offers new experiences for users.

4.3.8 Sensor Registry

The Sensor Registry contains a list of *sensorIds*, their names, their *property* options offered, and the URL of the SOS Server at which their real-time data can be retrieved. The Sensor Registry provides a set of services to the Application Server for querying the available sensors, which the Application Server can then send to the client for display in a list. It was implemented as an independent JBoss application as shown in Figure 4.23. The *SensorRegistryServlet* class is queried by the Application Server for the list of available real-time Sensors that is to be displayed to the client. A list of SOS Servers is obtained by scheduling a *SensorNetworkFinder* to use various Sensor Network discovery techniques at a regular interval. These techniques can include checking a local XML-based settings file of all known public SOS-compliant Sensor Servers or performing a HTTP request for public lists such as via Google (as mentioned in Section 4.3.7). Once a list of SOS Servers is obtained, the SOS Servers are queried by using the methods defined by the SOS standard. More specifically, a *GISCapabilitiesReader* object is created to perform a *GetCapabilities* request and obtain a list of individual *sensorIds*, sensor

names, and their available *observedProperty* options. This information is returned to the Application Server along with the URL of the SOS Server, all of which is sent to the client when the list of Sensors is requested from the user interface. When a client subscribes to a specific Sensor by forming a Custom XML message, the Smart Publisher is given enough information to be able to contact the SOS Server and retrieve the real-time data for the requested Sensor.

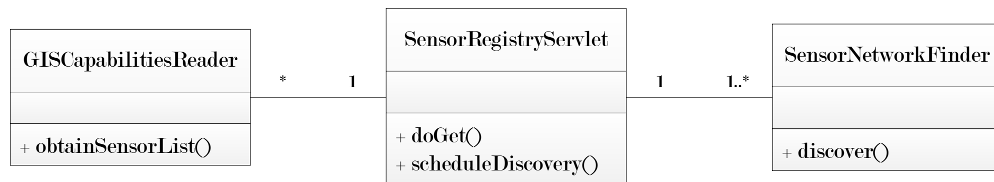

FIGURE 4.23: Sensor Registry Class Diagram.

Figure 4.24 summarizes the interactions between the Client (GIS Web Application and Collaborative Web Client Platform), the Application Server (including the Subscriber component), the Sensor Registry, the Smart Publisher and the JMS Server.

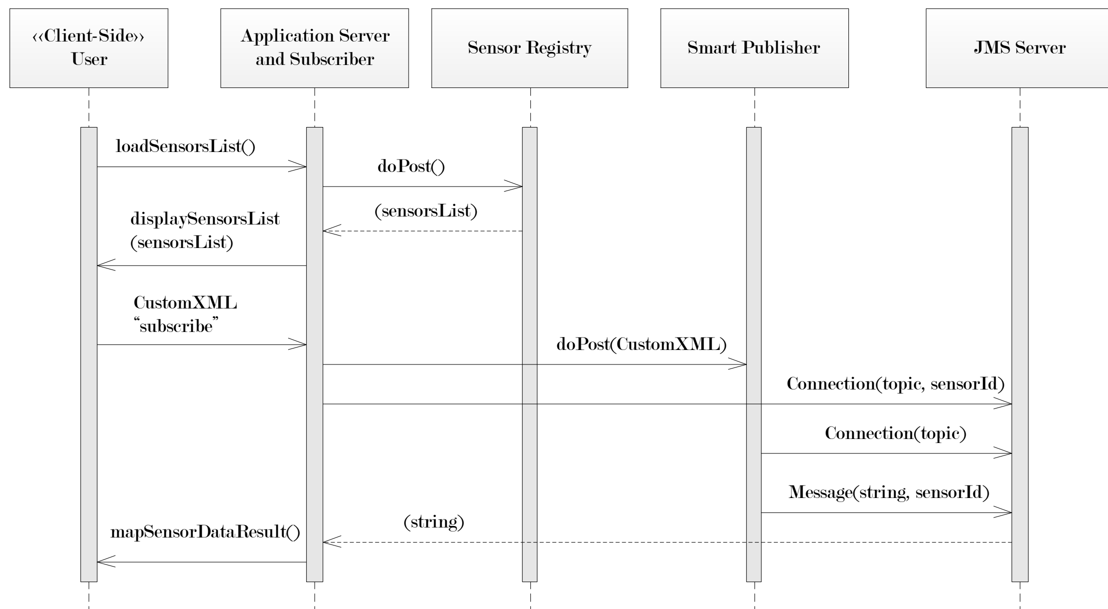

FIGURE 4.24: Sensor Registry Message Sequence Chart.

Chapter 5

Results

The architecture developed was tested in a variety of ways to observe its performance in showing real-time sensor data within collaborative web-based environments. Before displaying real-time sensor data, the two in-house collaborative platforms (UC-IC and Watch Together) had to be deployed and evaluated for their real-time collaboration abilities. Testing was performed on a variety of desktop PCs, laptops and mobile devices. The client-side browser was typically Firefox running on Intel Core 2 Duo machines with 4GB of RAM in Windows, while the server-side components were deployed to 2.2GHz AMD Opteron machines with 2GB of RAM running Ubuntu. Once real-time collaboration among multiple users was found to be successful, the GIS Web Application of each collaborative environment could be tested with different sensor data sources. This chapter presents the results that were obtained when evaluating the various components of the architecture from the point-of-view of a user within an online collaborative environment.

5.1 UC-IC Results

This section will first present the basic collaboration scenarios that were tested with UC-IC to demonstrate the real-time nature of the environment. Before attempting to work with real-time GIS data, several tests were also performed with archived sensor data stored in the Spatial Database component of the architecture.

5.1.1 Basic Collaboration

The resulting user interface for UC-IC, as detailed in Section 4.1.2, can be seen in Figure 5.1. It consists of an online desktop-like environment where applications appear in separate windows that can be resized and repositioned around the workspace. A group of icons representing a user's contacts appears along the right side of the user interface. The icons light up to indicate that the user is online and that applications can be dragged to this user to request a collaborative session.

FIGURE 5.1: UC-IC Collaborative Environment.

In order to evaluate the collaborative functionality of the UC-IC environment, the UC-IC Server was deployed within an instance of JBoss running on an internal Linux-based server. User accounts and permissions for multiple test users were configured via the UC-IC Administrator account. Multiple users using different computers then connected to the UC-IC Server and logged in with the given accounts (multiple web browser instances on the same computer can also be used for testing but here it is assumed that just one web browser is running per computer for clarity). They were then able to open various UC-IC applications and send them to other users for collaboration.

Figure 5.2 shows seven different users taking part in a UC-IC collaborative session for a photo viewer application. A user with a Nokia N95 smartphone is using the web browser on the phone to log in to the UC-IC environment and share a photo of a nearby plant. The user had just taken this photo by using the camera built into the phone. The other users (three using laptops, two using desktops, and one using the browser on an iPhone) have accepted the photo viewer application

from the Nokia N95 user. A user may now annotate the photograph (by drawing a red circle to highlight something, for example) and all other users will instantly receive the update. Note how all receiving users had some of the other available UC-IC applications open (maps in JIP, calculator, rich-text editor etc.) before having accepted the photo viewer application.

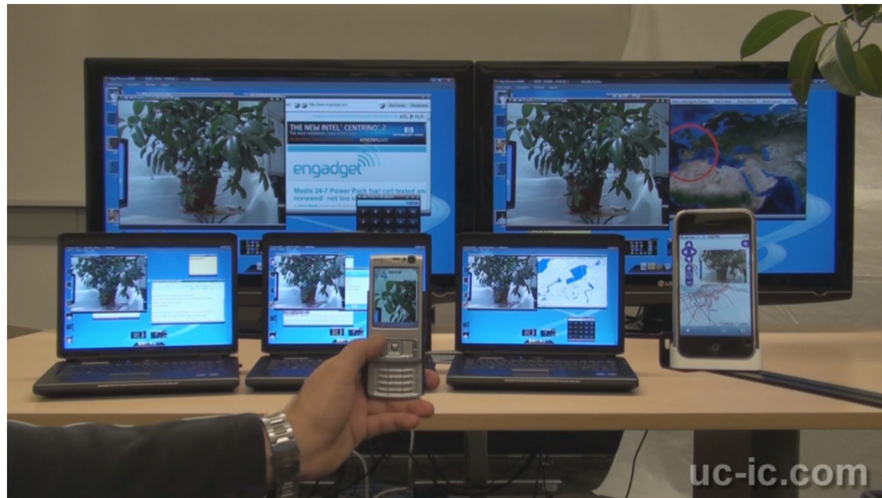

FIGURE 5.2: UC-IC Photo Viewer Collaborative Session.

5.1.2 Basic GIS Collaboration

Since real-time collaboration for various UC-IC applications proved to be quick and reliable, the remaining evaluations focused on the JIP mapping application that was developed for UC-IC. As described in Section 4.1.3, the JIP application consists of a *JIP Search Window*, *JIP Map Window*, *JIP Layers Window*, and *JIP Info Window*. When all four windows are present, the application appears as shown in Figure 5.3. A search term entered into the *JIP Search Window* will update the location of the *JIP Map Window*. The *JIP Map Window* lets the user explore the location through panning and zooming. The *JIP Layers Window* offers a choice of public base layers, including the "OpenStreetMap WMS" layer which contains roads (the OpenStreetMaps project encourages users to submit GPS data for the streets in their city to build an open database [82]), and the "NASA Global Mosaic" layer, which contains satellite images. Below the layers list is a list of real-time sensors which the user can subscribe to. Finally, the *JIP Info Window* is present for displaying additional information when the user clicks a marker on the *JIP Map Window*.

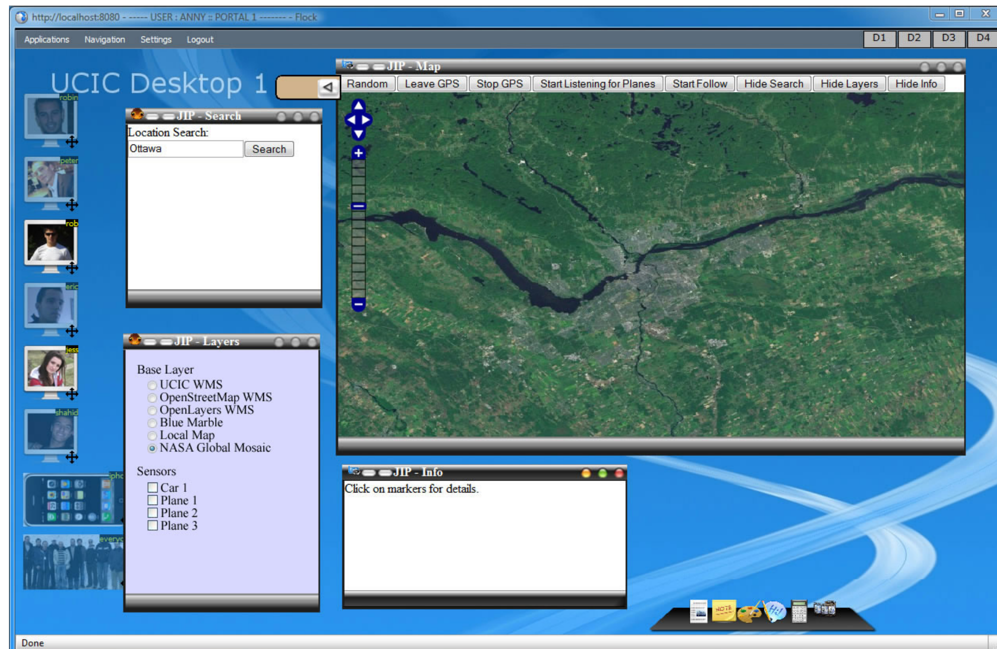

FIGURE 5.3: UC-IC Environment with JIP Windows Open.

A collaborative map session was started by dragging the *JIP Map Window* to another online user. Figure 5.4 shows how two users on two different computers are in a collaborative session with JIP. A red circle drawn by one user appears on the collaborative *JIP Map Window* of the other user. A close-up of each screen is provided in the top half of the figure.

FIGURE 5.4: Two Users Collaborating on a JIP Map Window.

Date	Time	Latitude	Longitude	Temperature (°C)
2010-30-04	01:00:00	40.7833	-73.9667	22
2010-30-04	01:00:00	40.6392	-73.7789	21
2010-30-04	01:00:00	40.7772	-73.8725	24
2010-30-04	01:00:00	40.85	-74.0608	23
2010-30-04	06:00:00	40.7833	-73.9667	21
2010-30-04	06:00:00	40.6392	-73.7789	21
2010-30-04	06:00:00	40.7772	-73.8725	23
2010-30-04	06:00:00	40.85	-74.0608	20
2010-30-04	11:00:00	40.7833	-73.9667	20
2010-30-04	11:00:00	40.6392	-73.7789	19
2010-30-04	11:00:00	40.7772	-73.8725	22
2010-30-04	11:00:00	40.85	-74.0608	19
2010-30-04	16:00:00	40.7833	-73.9667	25
2010-30-04	16:00:00	40.6392	-73.7789	26
2010-30-04	16:00:00	40.7772	-73.8725	28
2010-30-04	16:00:00	40.85	-74.0608	27
2010-30-04	21:00:00	40.7833	-73.9667	25
2010-30-04	21:00:00	40.6392	-73.7789	24
2010-30-04	21:00:00	40.7772	-73.8725	26
2010-30-04	21:00:00	40.85	-74.0608	27

Table 5.1: Sensor Data Values from Four Locations.

5.1.3 General GIS Data

To test the ability of the JIP application to access sensor data, the display of static sensor data within the *JIP Map Window* was tested first. The data shown in Table 5.1 (obtained from NOAA for the New York area) was manually added to the SOS Server by using the correct *InsertObservation* requests. The SOS Server stored this data within its PostgreSQL Spatial Database component for later retrieval. A button was added to the *JIP Map Window* to query the Sensor Server directly from the client by sending a *GetObservation* request in the SOS format and parsing the server's response. The response was then displayed within the *JIP Info Window* as a graph by passing the parsed Sensor Server response data to the open source JFreeChart API [90]. The resulting graph containing the archived GIS data is shown in Figure 5.5.

A second test was performed with the static sensor data in which data was retrieved directly from the SOS Server for display as markers within the *JIP Map Window*. A simple rules-based colouring scheme was then applied to help the user to better

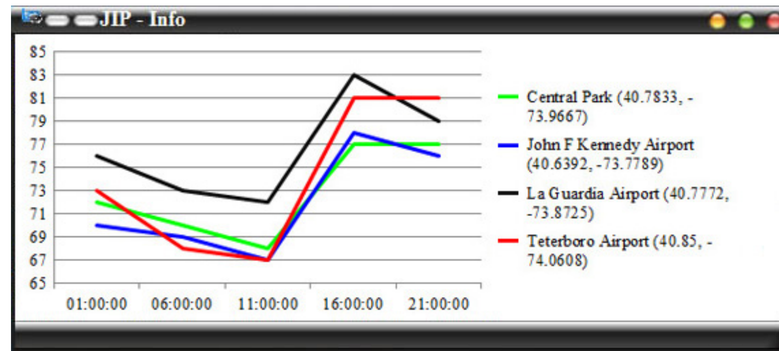

FIGURE 5.5: Graph of Sensor Data Values from Table 5.1.

visualize the range of temperature values once the values are plotted on a web-based map. The resulting map (using the default OpenLayers base layer and data at time 11:00:00) can be seen in Figure 5.6. The JIP application was therefore successfully able to retrieve static sensor data and plot it within a collaborative web-based environment.

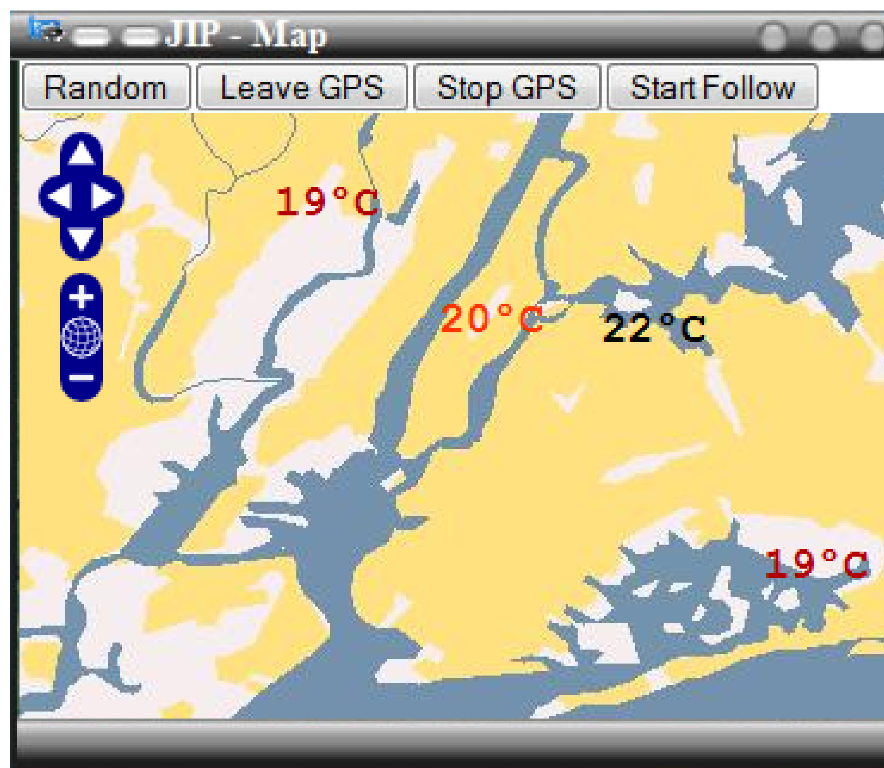

FIGURE 5.6: Rules-Based Colouring of Sensor Data from Table 5.1.

5.2 Watch Together Results

Two deployments of Watch Together were used to ensure the platform provides a quality real-time collaboration experience. The first was a deployment as a Facebook application to test and observe how real-world users interact with collaborative applications. The second was an internal deployment which offered more flexibility for testing the Google Maps application.

5.2.1 Facebook Deployment

In order to observe how real users interact with synchronized applications and to truly test the quality of the implementation of the Collaborative Web Client Platforms, Watch Together was deployed to the public as a Facebook Application [83]. Users can access the system by logging in with their Facebook account and adding Watch Together to their application bookmarks list. Facebook's API [91] was used to retrieve the information about the currently logged in user (such as the user's name, their list of friends, their profile image, etc.) and to populate the user interface.

Figure 5.7 shows six users collaboratively watching a YouTube video. The Viewer Module described in Section 4.2.2 appears in the top half of Watch Together and is always synchronized between the users in the session. The users currently in the session are shown along the bottom of the Watch Together interface using either their profile images (retrieved from the user database) or a live video stream from the user's webcam.

Near the middle of the Watch Together interface is a menu bar with clickable icons. The left side of the menu bar contains the list of applications developed for the Watch Together platform (by using the API described in section 4.2.2) to support various sources of popular online multimedia content. Clicking one of these icons brings up a search module containing thumbnails of the search results. The thumbnails can be clicked on to change what is displayed within the viewer module. The right side of the menu contains icons for the contacts module (which shows a list of available online users that can be invited to a session), text chat module, and the settings module, all of which appear on top of the viewer module but do no affect the synchronized content. The user's latency relative to the RTMP server is displayed beside their name in the bottom left corner.

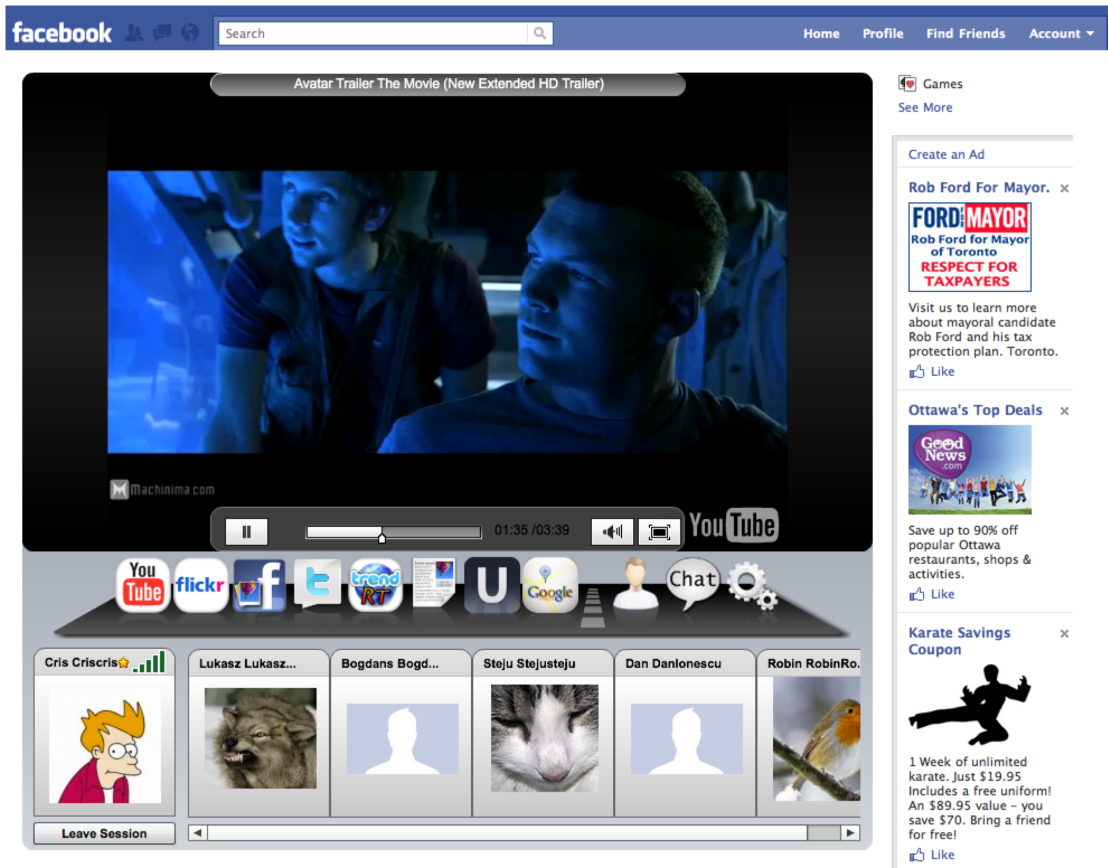

FIGURE 5.7: Six Facebook Users Collaboratively Watching a YouTube Video.

This deployment was used in order to determine the response of users to the collaborative nature of the application. Based on over a thousand users who opted in to share anonymized usage data, 65% of users of Watch Together were found to be male and 35% were female. Surprisingly, 24% of users enabled their webcam when using Watch Together, which is a strong indicator that users enjoy sharing and discussing online content in this collaborative fashion. The YouTube application was the most popular, with an average of 11.4 videos viewed per user. As a Collaborative Web Client Platform, Watch Together is therefore proven to be robust and easy-to-use, with many real people returning to use the Facebook application daily.

5.2.2 Internal Test Deployment

The Facebook deployment of Watch Together proved that the collaboration capabilities of the platform are good enough for real-world use. For further testing, an internal deployment of the system was connected to a test user database with a

simple login system developed using Java Server Faces (JSF). A webpage presented the user with a username and password login screen, and a successful login would load a page that had the compiled Watch Together SWF embedded within it. Figure 5.8 shows three users participating in a video chat session with the Google Maps application on this JEE-based test deployment. The system properly kept all content synchronized among the users, and, at the same time, it allowed them to use video chat to discuss the content.

FIGURE 5.8: Map Sharing and Video Chat.

As described in Section 4.2.3, the Google Maps application was created by using the MediaAPI and consists of a *Search Module*, a *Viewer Module*, a *User Control Module* and an *Information Module*. The *User Control Module* appears on top of the *Viewer Module* and is shown is shown in Figure 5.9. It contains the list of Sensors to which the user may subscribe based on the information retrieved by the Application Server from the Sensor Registry.

5.3 Real-Time GIS Data

Once the basic map application functionality within the two collaborative platforms was established to be behaving as intended, three cases were explored that

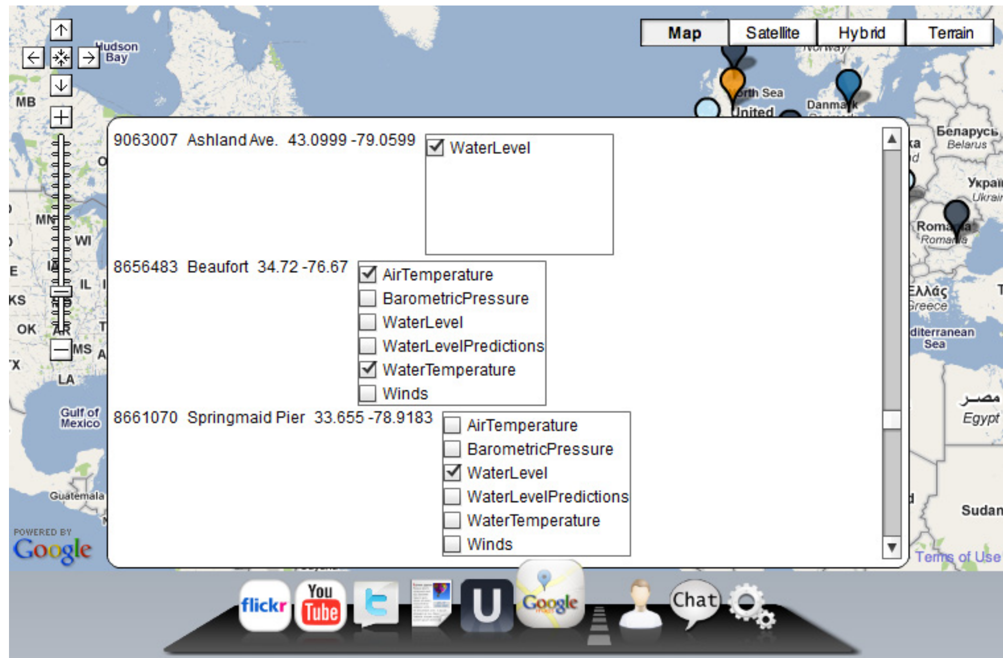

FIGURE 5.9: Watch Together Map Application Sensors List.

involved the display of a variety of real-time sensor data within collaborative environments.

5.3.1 Public Sensor Data

As described in Section 4.3.8, the Sensor Registry uses various methods for discovering Sensor Networks of existing Sensors that are freely available on the Internet. One such source of sensors that was discovered is the SOS Server from NOAA [27]. The NOAA SOS Server reports on over a hundred sensors with real-time data such as temperature from various weather stations across the United States. The Sensor Registry was able to discover all offered sensors and make them available to the GIS Web Applications when requested by the Application Server. Users of the GIS Web Applications could then subscribe to any number of the sensors to receive their real-time updates. When a subscription is requested, the internal Smart Publisher is tasked with polling NOAA's external SOS Server for the latest sensor data and asynchronously sending this data to all interested subscribers through the internal JMS Server. Figure 5.10 shows how the real-time data appears within the Google Maps application of Watch Together when a marker is selected. The selected sensors are synchronized to all other contacts within the collaborative session.

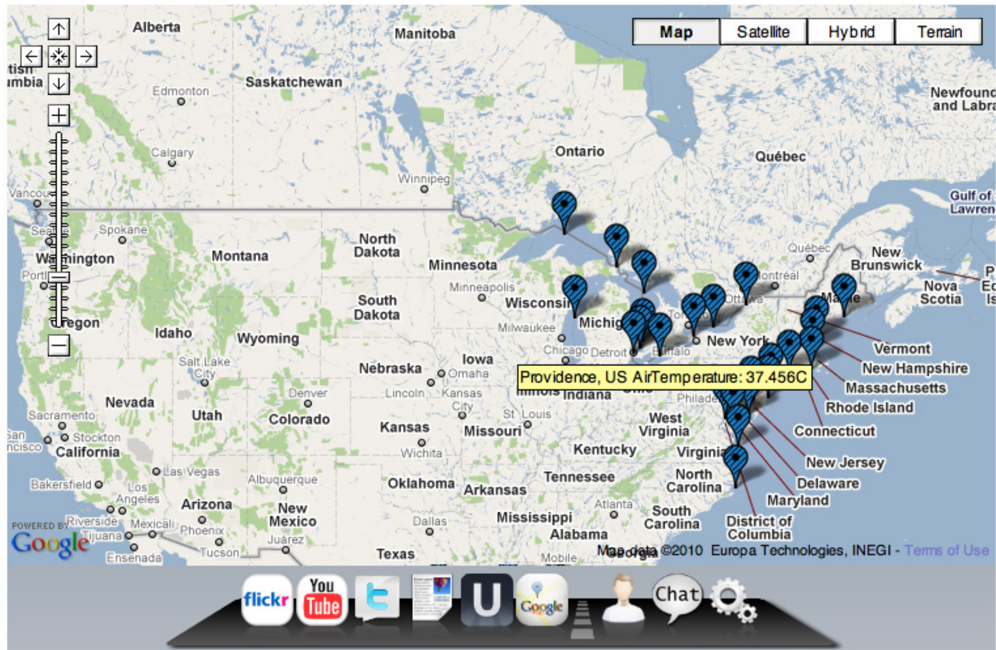

FIGURE 5.10: NOAA Sensors in Google Maps Application.

5.3.2 Vehicle GPS Data

Smartphones like the Nokia N95 contain a GPS chip that returns the current latitude and longitude information of the phone with decent accuracy. This data is accessible from a native application running on the phone by using Nokia's API [92]. A mobile version of the Sensor Feeder was implemented for the Nokia device, allowing it to transmit its latest GPS coordinates to an internal Sensor Server. Since the phone has data connectivity within a large part of Ottawa through the Rogers network, a user was able to drive around some parts of the city and broadcast their latest GPS position to the SOS Server. Since all sensors in the internal SOS Server are gathered by the Sensor Registry, the phone appeared as a sensor that users could subscribe to via the GIS Web Applications. Their real-time movement could then be observed from the JIP application, as shown in Figure 5.11. In addition, Figure 5.12 shows how two users (one on a laptop and one on an iPhone) are able to collaborate over the same real-time data within their web browsers (note the blue car in each; the red circles were added to the image and are not part of the collaborative session as was shown in Figure 5.4). The marker representing the car would move to reflect the latest GPS location obtained by the Smart Publisher from the SOS Server and sent to the GIS Web Application through the JMS Server. Clicking the marker would reveal the numerical latitude and longitude values within each application's information area, as shown in

Figure 5.13.

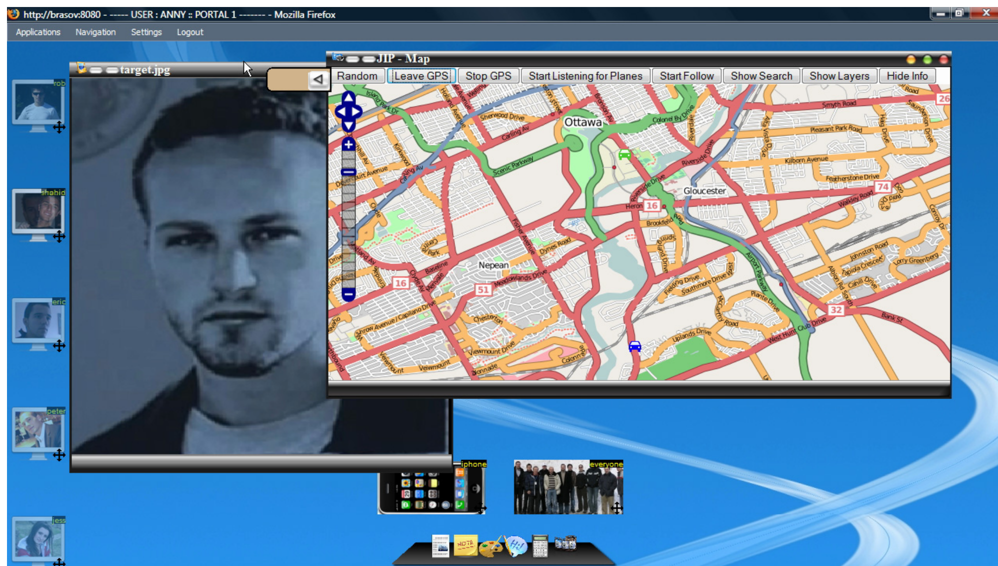

FIGURE 5.11: Real-Time GPS-Based Vehicle Movement in JIP.

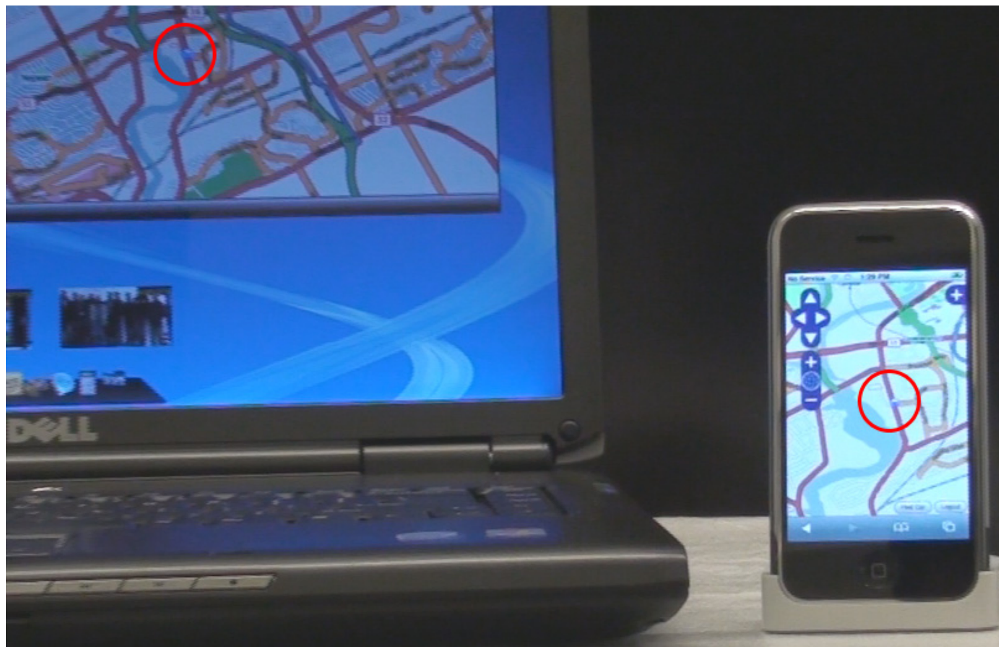

FIGURE 5.12: Real-Time Vehicle Tracking From Desktop and iPhone.

Unfortunately, the unreliable nature of cell phone data networks and limited processing power of the N95 phone made it difficult to judge the latency between the phone's actual position and the latest position displayed on the web-based map. The delay was estimated at about three seconds. A different approach needed to be taken if more accurate measurements were to be observed.

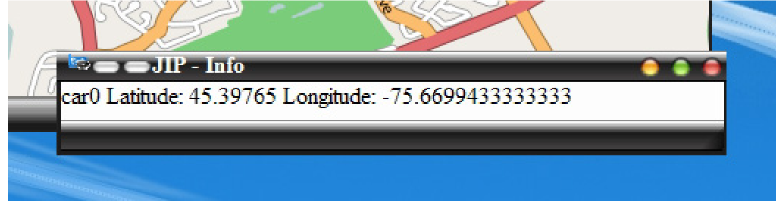

FIGURE 5.13: Real-Time Latitude/Longitude Data in JIP Info Window.

5.3.3 Aircraft Data

To properly evaluate how the real-time architecture functions within a collaborative web-based environment, a controllable real-time data source was needed. Microsoft Flight Simulator 2004 was selected since the real-time aircraft data from inside the game could be accessed through a driver called FSUIPC. In this case, the Sensor is actually a memory location in which the video game stores its variables. By setting up the Sensor Feeder with the driver to obtain real-time data from the game, the data could be sent to the SOS Server and asynchronously reach the GIS Web Applications.

The system was tested with three different Flight Simulator instances running on three different computers. With the Sensor Feeder sending the real-time data to the Sensor Server, and the Sensor Registry providing the list of Sensors to users of the GIS Web Application, the airplane data could be subscribed to from within the collaborative environments. A subscription would activate the Smart Publisher to begin requesting data from the SOS Server. The Smart Publisher would then send the data to the JMS Server for distribution via the Application Server to the subscribed clients.

Figure 5.14 shows how the three planes were drawn as markers within JIP (top), as well as screenshots of the three Flight Simulator instances at that moment in time (bottom). In this case, the data typically displayed within the *JIP Info Window* was placed into the *JIP Map Window* to save space. As the planes moved, a marker within the *JIP Map Window* would move to show each plane's latest location. In addition, clicking a marker would update the displayed data with additional streaming information, such as the plane's current altitude, air speed and heading. Since the game provides a multiplayer feature, the three planes were actually in the same game environment (in fact, with a closer look, a distant plane can be seen in the windshield of the plane shown in the bottom left corner of the figure).

FIGURE 5.14: Real-Time Flight Simulator Data Inside JIP.

One of sixteen possible icons representing the proper orientation of each plane was loaded using a rules-based approach based on the heading of the plane. In addition, the plane maker would change colour to red in the case of a simulated engine failure, which could be quickly activated from within Flight Simulator. This is shown in Figure 5.15. The delay between the time the engine failure was triggered in Flight Simulator to the time it appeared within the web application was timed to be 0.33 seconds when testing on a local network. This value was obtained by recording a video containing all PCs such that the exact time of both changes can be observed. This value is less than other web-based real-time solutions that have been discovered which were also testing on local networks [43].

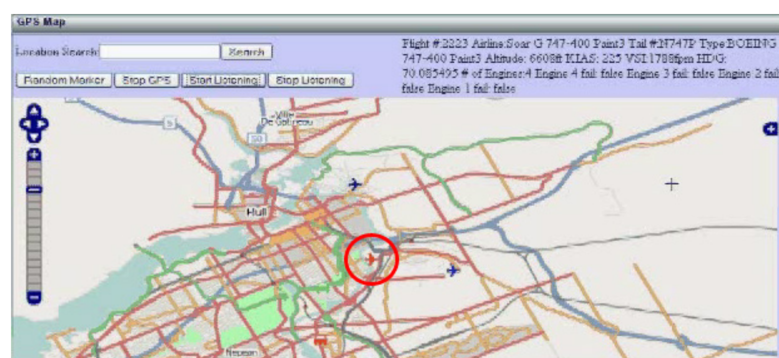

FIGURE 5.15: Real-Time Marker Update from Engine Failure.

In UC-IC, the JIP Map Window was shared with five other UC-IC users who could navigate and draw on the map while the planes were moving. In Watch Together, users could easily be invited to a collaborative session where the planes were being watched and their progress discussed over webcam chat.

Chapter 6

Conclusion

The rising popularity of social networking websites like Twitter and Foursquare shows that the demand for real-time data on the internet is now higher than ever before. Increasingly, users are using the web for distributing and consuming real-time data such as location information. While existing attempts at distributing real-time sensor data show potential, the lack of a standardized architecture for supporting a variety of data and collaboration needs is preventing the widespread availability of real-time GIS data on the web. By using the latest web-based technologies, real-time communication techniques, and sensor data standards, this thesis presents an adaptable architecture that allows real-time sensor data to be transmitted to and retrieved from collaborative web-based environments.

6.1 Concepts Addressed in this Thesis

The research presented in this thesis examines the creation of a standard architecture for decreasing the development and deployment time of systems that require real-time GIS data to be accessible from a typical web browser. The architecture defines the components, protocols, and guidelines required for setting up a reliable and accessible real-time sensor data distribution service where the data can be requested and monitored from within a web browser. While the concept of real-time data delivery to a web browser is not new in literature, the novelty of this thesis comes in the form of using open standards, protocols, and established techniques such as publisher/subscriber messaging in order to provide for optimal real-time performance. This thesis also builds on recent online trends in social networking

and browser-based collaboration to ensure the real-time data is presented in an accessible, user-friendly, and useful manner.

This thesis also presents an implementation of the architecture by using two different collaborative platforms to access a large variety of sensor data sources. The collaboration techniques and APIs of the platforms are also given. The implementation uses the open Sensor Observation Service (SOS) standard to manage the sensor data and ensures it reaches interested clients in real-time through the publisher/subscriber push-based method. The specifications for all components, as well as the details of the communication between them, are presented. The implementation is evaluated using real-time data originating from a flight simulation game such that accurate delay measurements can be made. The real-time sensor data is also shared with other users of each collaborative environment.

6.2 Contributions of this Thesis

This thesis examines the creation of a publisher/subscriber-based architecture for delivering real-time sensor data to users collaborating from practically any web-enabled device. Open standards and data protocols are used to ensure interoperability between the components of the architecture.

More specifically, the need for a standardized architecture and the issues that such an architecture must deal with are presented. The need was demonstrated through a review of existing literature for both real-time web GIS and real-time web collaboration problems, as well as a review of real-time web architectures and standards that deal with aspects of real-time messaging. An adaptable architecture which makes use of proven standards is created in order to address these issues. Mechanisms that ensure the support for practically all GIS-related real-time data sources are presented. In addition, methods for discovering other existing standards-compliant sensor networks are given. The design also addresses issues of sensor data persistence as well as the need for a registry to store all known sensors. Two different collaborative platforms are implemented based on the requirements of the architecture. A map application built using the API of each environment allows users to subscribe to the real-time sensors of their choice and to invite other online contacts to a collaborative session where all users see the same map and sensor state.

This thesis presented a number of significant research contributions which can be summarized as follows:

1. The components, protocols and best practices of a publisher/subscriber-based architecture is presented. The architecture makes it possible to deliver real-time sensor data from a raw sensor source to a collaborative web application accessible from within a web browser. Through the creation of a standard architecture which consists of eleven main components with clearly-defined roles, the time to develop new real-time data distribution systems is decreased as integrators can focus on their specific requirements.

2. A mechanism based on open GIS standards is given which allows the integration of a large variety of real-time sensors into the system. By adapting the specific format of a sensor to the established Sensor Observation Service standard, the data can be submitted to an SOS server to be indexed by a spatial database and made available to interested clients through well-defined REST-based request/response message exchanges. Methods for discovering and integrating other public sources of SOS data are also presented.

3. The AJAX-based and server-side implementation of the collaborative functionality of the UC-IC real-time platform for SIP-based multi-domain collaboration is presented, as well as the implementation of the Joint Intelligence Picture (JIP) application for flexible multi-instance organization of GIS data. UC-IC uses AJAX to allow users to send, share and distribute control over applications and their data in real-time, making it an ideal platform and environment for productivity-related collaboration involving real-time sensor data.

4. The Flash-based and server-side implementation of the real-time platform for collaborative multimedia known as Watch Together is presented. Watch Together achieves real-time collaboration between the different clients in the same session, while at the same time is extendable with new multimedia types through its MediaAPI. The implementation of a Google Maps-based map application is given that takes advantage of the animation and webcam streaming capabilities of Flash.

5. A mechanism for simulating real-time data by using the driver of a flight simulator game as a sensor is given. Real-time data is shown to stream smoothly

to both client applications and become available for real-time collaboration with other users of each respective platform.

6.3 Future Research

While this thesis presented an architecture that allows for adapting to many different hardware and software systems from different vendors, the implementation of a large number of these systems is outside the scope of this thesis. As such, more components need to be developed in order to further validate the architecture and ensure that it can properly integrate with other sensor standards (such as RTCM), publisher/subscriber mechanisms (such as XMPP), and collaborative platforms (such as Google Wave).

Another avenue for research is that of two-way communication with the real-time data source. Rather than passively receiving data, a client application can be made to instruct sensors in real-time to reconfigure options such as their sampling frequencies. This would be useful in cases where a sensor's accuracy needs to be improved in anticipation of significant events or to save on operational costs when no events of interest are present. Models can be used to develop adaptive sensing strategies for such cases.

Bibliography

[1] (2010) Twitter. Twitter Inc. [Accessed: December 2010]. [Online]. Available: http://www.twitter.com

[2] L. Rao. (2010, October) Twitter Added 30 Million Users In The Past Two Months. TechCrunch. [Accessed: December 2010]. [Online]. Available: http://techcrunch.com/2010/10/31/twitter-users/

[3] (2010, December) Foursquare Hits 2 Million Check-ins, 25K New Users Daily. TechCrunch. [Accessed: December 2010]. [Online]. Available: http://techcrunch.com/2010/12/08/foursquare-hits-2-million-check-ins-25k-new-users-daily/

[4] P. M. (2010) HousingMaps. [Accessed: December 2010]. [Online]. Available: http://www.housingmaps.com/

[5] D. A. Norman, "The Way I See It - Systems Thinking: A Product is More Than the Product," *Interactions*, vol. 16, no. 5, pp. 52–54, 2009.

[6] D. Sarno. (2009, August) L.A., SoCal street traffic now visible on Google Maps. L.A. Times. [Accessed: December 2010]. [Online]. Available: http://latimesblogs.latimes.com/technology/2009/08/la-socal-street-traffic-now-visible-on-google-maps.html

[7] (2009) Google Latitude. Google Inc. [Accessed: December 2010]. [Online]. Available: http://www.google.com/intl/en_us/latitude/intro.html

[8] (2009, February) Google Latitude service. Google Inc. [Accessed: December 2010]. [Online]. Available: http://www.google.com/support/forum/p/maps/thread?tid=066870485694ec49%&hl=en

[9] (2010) Live Ships Map - Automatic Idenfitication System - Vessel Traffic and Positions. University of the Aegean. [Accessed: December 2010]. [Online]. Available: http://www.marinetraffic.com

[10] (2010) Adobe Labs - Adobe Flash Platform Technologies. Adobe Systems Inc. [Accessed: December 2010]. [Online]. Available: http://labs.adobe.com/technologies/flash/

[11] B. Mller, "An FCCC Impact Response Instrument as part of a Balanced Global Climate Change Regime," *Oxford Institute for Energy Studies*, 2002.

[12] (2010) LINET data by Nowcast. Nowcast. [Accessed: December 2010]. [Online]. Available: https://www.nowcast.de/en/produkte-und-vorteile/linet-data.html

[13] M. R. Thissen, J. M. Page, M. C. Bharathi, and T. L. Austin, "Communication Tools for Distributed Software Development Teams," in *SIGMIS-CPR '07: Proc. of ACM SIGMIS CPR Conf. on Computer Personnel Research*. New York, NY, USA: ACM, 2007, pp. 28–35.

[14] (2010) Google Docs - Online Documents, Spreadsheets, Presentations. Google Inc. [Accessed: December 2010]. [Online]. Available: http://docs.google.com/

[15] Welcome to the OGC Website. Open Geospatial Consortium Inc. [Accessed: December 2010]. [Online]. Available: http://www.opengeospatial.org/

[16] (2003) Flight Simulator 2004: A Century of Flight. Microsoft Corp. [Accessed: December 2010]. [Online]. Available: http://www.microsoft.com/games/pc/flightsimulator.aspx

[17] (2010) Flex Open-Source Framework. Adobe Systems Inc. [Accessed: December 2010]. [Online]. Available: http://www.adobe.com/products/flex/

[18] (2010) Real-Time Messaging Protocol (RTMP) Specification. Adobe Systems Inc. [Accessed: December 2010]. [Online]. Available: http://www.adobe.com/devnet/rtmp.html

[19] (2010, June) Flash Player Version Penetration. Adobe Systems Inc. [Accessed: December 2010]. [Online]. Available: http://www.adobe.com/products/player_census/flashplayer/version_penetration.html

[20] R. Dagher, C. Gadea, B. Ionescu, D. Ionescu, and R. Tropper, "A SIP Based P2P Architecture for Social Networking Multimedia," in *DS-RT 2008: 12th IEEE/ACM Int. Symp. on Distributed Simulation and Real-Time Applications*. IEEE Computer Society, October 2008, pp. 187–193.

[21] R. Tropper, R. Dagher, C. Gadea, B. Ionescu, and D. Ionescu, "UC-IC: A Cloud Based and Real-Time Collaboration Platform Using Many-to-Many-on-Many Relationship," in *Proc. of 8th IEEE I2TS*. IEEE Computer Society, May 2009, pp. 9–17.

[22] C. Gadea, B. Ionescu, and D. Ionescu, "Real-Time Collaborative Intelligent Services for Sensor Networks," in *ICCC-CONTI: 2010 Int. Joint Conf. on Computational Cybernetics and Technical Informatics*. IEEE Computer Society, May 2010, pp. 511–516.

[23] D. Ionescu, C. Gadea, and B. Ionescu, "Collaborative Web-Based Architecture for Real-Time Monitoring of Sensor Data," in *EESMS 2010: Proc. of IEEE Workshop on Environmental, Energy, and Structural Monitoring Systems*. IEEE Computer Society, September 2010.

[24] C. Gadea, D. Ionescu, and B. Ionescu, "Collaborative Web-Based Mapping of Real-Time Flight Simulator and Sensor Data," in *OSGeo Journal*, vol. 7. OSGeo, December 2010, in press.

[25] C. Gadea, B. Solomon, B. Ionescu, and D. Ionescu, "A Real-Time Browser-Based Collaboration System for Synchronized Online Multimedia Sharing," in *Proc. of 9th IEEE I2TS*. IEEE Computer Society, May 2010.

[26] C. Gadea, B. Ionescu, D. Ionescu, S. Islam, and B. Solomon, "A Distributed Online Environment for Gesture-Based Collaboration," in *Proc. of 9th IEEE I2TS*. IEEE Computer Society, May 2010.

[27] (2010) CO-OPS' Implementation of IOOS Sensor Observation Service (SOS). National Oceanic and Atmospheric Administration's Center for Operational Oceanographic Products & Services. [Accessed: December 2010]. [Online]. Available: http://opendap.co-ops.nos.noaa.gov/ioos-dif-sos-test/

[28] W. Howe. (2010, March) A Brief History of the Internet. [Accessed: December 2010]. [Online]. Available: http://www.walthowe.com/navnet/history.html

[29] P. Graham. (2005, November) Web 2.0. [Accessed: December 2010]. [Online]. Available: http://www.paulgraham.com/web20.html

[30] T. M. Thomas, *Java Data Access: JDBC, JNDI, and JAXP*. Hungry Minds, 2001.

[31] (2010) World Wide Web Consortium (W3C). W3C. [Accessed: December 2010]. [Online]. Available: http://www.w3.org/

[32] (2010) MapQuest Maps - Driving Directions - Map. MapQuest Inc. [Accessed: December 2010]. [Online]. Available: http://www.mapquest.com/

[33] N. C. Zakas, J. McPeak, and J. Fawcett, *Professional Ajax, 2nd Edition (Programmer to Programmer)*. Wrox, 2007.

[34] (2010) Google Maps. Google Inc. [Accessed: December 2010]. [Online]. Available: http://maps.google.com/

[35] (2005, June) Definition: Slippy Map. Fantom Planet. [Accessed: December 2010]. [Online]. Available: http://fantomplanet.wordpress.com/2005/06/23/definition-slippy-map/

[36] E. Schonfeld. (2007, October) Exclusive: MapQuest Plays Catch-Up With Launch of Beta. TechCrunch. [Accessed: December 2010]. [Online]. Available: http://www.techcrunch.com/2007/10/12/exclusive-mapquest-plays-catch-up-with-launch-of-beta/

[37] M. Jones. (2008, May) Introducing the Google Maps API for Flash. Google Inc. [Accessed: December 2010]. [Online]. Available: http://googlemapsapi.blogspot.com/2008/05/introducing-google-maps-api-for-flash.html

[38] (2010) ESRI - The GIS Software Leader - Mapping Software and Data. Environmental Systems Research Institute Inc. [Accessed: December 2010]. [Online]. Available: http://www.esri.com

[39] J. E. Harmon and S. J. Anderson, *The Design and Implementation of Geographic Information Systems*. Wiley, 2003.

[40] (2009, July) Commissioner's Findings - PIPEDA Case Summary #2009-008: Report of Findings: CIPPIC v. Facebook Inc. Office of the Privacy Commissioner of Canada. [Accessed: December 2010]. [Online]. Available: http://www.priv.gc.ca/cf-dc/2009/2009_008_0716_e.cfm

[41] D. Sacks. (2007, May) The New Portals: It's the Bread, Not the Peanut Butter. TechCrunch. [Accessed: December 2010]. [Online]. Available: http://www.techcrunch.com/2007/05/31/the-new-portals-its-the-bread-not-the-peanut-butter/

[42] G. Weber, D. Dettmering, and H. Gebhard, "Networked Transport of RTCM via Internet Protocol (NTRIP)," in *A Window on the Future of Geodesy*, ser. International Association of Geodesy Symposia, F. Sanso, Ed. Springer Berlin Heidelberg, 2005, vol. 128, pp. 60–64.

[43] Y. Heo, S. Lim, and C. Rizos. (2010) A Web-Based Real-Time Delivery of Global Navigation Satellite System Data. School of Surveying and Spation Information Systems, The University of New South Wales. [Accessed: December 2010]. [Online]. Available: http://developers.facebook.com/docs/chat

[44] C. Mayer, B. Stollberg, and A. Zipf, "Providing Near Real-Time Traffic Information within Spatial Data Infrastructures." IEEE Computer Society, February 2009, pp. 104–111.

[45] M. Botts, G. Percivall, C. Reed, and J. Davidson, *OGC Sensor Web Enablement: Overview and High Level Architecture*, Open Geospatial Consortium Inc. Std., December 2007, [Accessed: December 2010]. [Online]. Available: http://portal.opengeospatial.org/files/?artifact_id=25562

[46] *OpenGIS Implementation Standard: Sensor Observation Service*, Open Geospatial Consortium Inc. Std., January 2006, [Accessed: December 2010]. [Online]. Available: http://portal.opengeospatial.org/files/?artifact_id=12846

[47] S. Cox, *Observations and Measurements Part 1 - Observation Schema*, Open Geospatial Consortium Inc. Std., December 2007, [Accessed: December 2010]. [Online]. Available: http://portal.opengeospatial.org/files/?artifact_id=22466

[48] *OpenGIS Geography Markup Language (GML) Encoding Standard*, Open Geospatial Consortium Inc. Std., August 2007, [Accessed: December 2010]. [Online]. Available: http://portal.opengeospatial.org/files/?artifact_id=20509

[49] E. Gamma, R. Helm, R. Johnson, and J. M. Vlissides, *Design Patterns: Elements of Reusable Object-Oriented Software.* Addison-Wesley Professional, 1994.

[50] T. Lethbridge and R. Laganiere, *Object-Oriented Software Engineering: Practical Software Development using UML and Java.* McGraw-Hill Science/Engineering/Math, 2002.

[51] K. Birman and T. Joseph, "Exploiting Virtual Synchrony in Distributed Systems," vol. 21, no. 5. New York, NY, USA: ACM, 1987, pp. 123–138.

[52] (2009) Welcome to 52North. 52North.org. [Accessed: December 2010]. [Online]. Available: http://52north.org/

[53] *OGC Sensor Alert Service Implementation Specification*, Open Geospatial Consortium Inc. Std., May 2007, [Accessed: December 2010]. [Online]. Available: http://portal.opengeospatial.org/files/?artifact_id=24780&version=1

[54] (2009) SAS. 52North.org. [Accessed: December 2010]. [Online]. Available: http://52north.org/index.php?option=com_content&view=category&layout=blog&id=27&Itemid=34

[55] P. Saint-Andre, "RFC3920: Extensible Messaging and Presence Protocol (XMPP): Core," Jabber Software Foundation, Tech. Rep., October 2004, [Accessed: December 2010]. [Online]. Available: http://tools.ietf.org/html/rfc3920

[56] (2010) http://developers.facebook.com/docs/chat. Facebook Inc. [Accessed: December 2010]. [Online]. Available: http://developers.facebook.com/docs/chat

[57] A. Baxter, J. Bekmann, D. Berlin, J. Gregorio, S. Lassen, and S. Thorogood, "Google Wave Federation Protocol Over XMPP," Google Inc., Tech. Rep., July 2009, [Accessed: December 2010]. [Online]. Available: http://wave-protocol.googlecode.com/hg/spec/federation/wavespec.html

[58] R. S. Quattlebaum. (2008, February) Mobile XMPP. [Accessed: December 2010]. [Online]. Available: http://www.deepdarc.com/2008/02/14/mobile-xmpp/

[59] *OpenGIS Sensor Event Service Interface Specification (proposed)*, Open Geospatial Consortium Inc. Std., October 2008, [Accessed: December 2010]. [Online]. Available: http://portal.opengeospatial.org/files/?artifact_id=29576

[60] Y. Tian, P. R. Houser, S. Hongbo, and S. V. Kumar, "Integrating Sensor Webs with Modeling and Data-Assimilation Applications: An SOA Implementation," in *IEEE Aerospace Conference 2008.* IEEE Computer Society, March 2008, pp. 1–7.

[61] (2009, July) Community Driven Open Source Middleware. - JBoss Community. JBoss. [Accessed: December 2010]. [Online]. Available: http://www.jboss.org/

[62] Y. Liu, D. Hill, A. Rodriguez, L. Marini, R. Kooper, J. Myers, X. Wu, and B. Minsker, "A New Framework for On-Demand Virtualization, Repurposing and Fusion of Heterogeneous Sensors," in *CTS '09: Int. Symp. on Collaborative Technologies and Systems.* IEEE Computer Society, May 2009, pp. 54–63.

[63] G. Brake and N. Smets, "Developing Adaptive Mobile Support for Crisis Response in Synthetic Task Environments," in *Proc. of the 2nd Int. Conf. on Usability and Internationalization*, ser. UI-HCII'07. Berlin, Heidelberg: Springer-Verlag, 2007, pp. 510–519.

[64] (2010) Google Wave - Communicate and Collaborate in Real-Time. Google Inc. [Accessed: December 2010]. [Online]. Available: http://wave.google.com

[65] (2009) Google Wave Preview. Google Inc. [Accessed: December 2010]. [Online]. Available: http://wave.google.com

[66] (2009) Google Docs. Google Inc. [Accessed: December 2010]. [Online]. Available: http://docs.google.com

[67] A. Imine, "Decentralized Concurrency Control for Real-Time Collaborative Editors," in *Proc. of the 8th Int. Conf. on New Technologies in Distributed Systems*, ser. NOTERE '08. New York, NY, USA: ACM, 2008, pp. 41:1–41:9.

[68] P. Krill. (2010, October) W3C: Hold Off on Deploying HTML5 in Websites. InfoWorld Inc. [Accessed: December

2010]. [Online]. Available: http://www.infoworld.com/d/developer-world/w3c-hold-html5-in-websites-041?page=0,0

[69] (2009, May) Official Google Blog: Went Walkabout. Brought back Google Wave. Google Inc. [Accessed: December 2010]. [Online]. Available: http://googleblog.blogspot.com/2009/05/went-walkabout-brought-back-google-wave.html

[70] U. Hlzle. (2010, August) Update on Google Wave. Google Inc. [Accessed: December 2010]. [Online]. Available: http://googleblog.blogspot.com/2010/08/update-on-google-wave.html

[71] (2010) PyGoWave Server. PyGoWave. [Accessed: December 2010]. [Online]. Available: http://www.pygowave.net/

[72] P. Geyer. (2008, January) Allan Padgett Interview - Adobe Tour Tracker. CyclingFans.com. [Accessed: December 2010]. [Online]. Available: http://www.cyclingfans.com/allan_padgett_adobe_tour_tracker_interview1.html

[73] D. C. Schmidt and C. O'Ryan, "Patterns and performance of distributed real-time and embedded publisher/subscriber architectures," vol. 66. New York, NY, USA: Elsevier Science Inc., June 2003, pp. 213–223.

[74] B. Xu, M. Rose, and Z. Lin, "A Novel Approach to Convert Single-user Applications into Collaborative Applications," in *CSCWD '06: 10th Int. Conf. on Computer Supported Cooperative Work in Design*. IEEE Computer Society, May 2006, pp. 1–5.

[75] R. Tropper, "New Architecture and Programming Paradigms for Cloud Collaborative RIA," MASc Thesis, School of Information Technology and Engineering, University of Ottawa, December 2009.

[76] J. Rosenberg, H. Schulzrinne, G. Camarillo, A. Johnston, J. Peterson, R. Sparks, M. Handley, and E. Schooler, "Rfc3621: Session initiation protocol," Tech. Rep., June 2002, [Accessed: December 2010]. [Online]. Available: http://www.ietf.org/rfc/rfc3261.txt

[77] W. Wang, "Powermeeting: GWT-Based Synchronous Groupware," in *HT '08: Proc. of 19th ACM Conf. on Hypertext and Hypermedia*. New York, NY, USA: ACM, 2008, pp. 251–252.

[78] (2010) JavaScript for Object-Oriented Programmers. Sun Microsystems. [Accessed: December 2010]. [Online]. Available: http://developers.sun.com/scripting/javascript/ajaxinaction/Ajax_in_Action_ApB.html

[79] E. Wolf and K. Howe, "Web-Client Based Distributed Generalization and Geoprocessing," in *GEOWS '09: Int. Conf. on Advanced Geographic Information Systems & Web Services.* IEEE Computer Society, February 2009, pp. 123–128.

[80] (2009, June) Welcome - GeoServer. GeoServer. [Accessed: December 2010]. [Online]. Available: http://www.geoserver.org

[81] L. Plesea. (2008, October) OnEarth, JPL WMS Server. National Aeronautics and Space Administration. [Accessed: December 2010]. [Online]. Available: http://onearth.jpl.nasa.gov/

[82] S. Coast. (2009) OpenStreetMap. OpenStreetMap Foundation. [Accessed: December 2010]. [Online]. Available: http://www.openstreetmap.org/

[83] (2009, June) Watch Together. [Accessed: December 2010]. [Online]. Available: http://www.watch-together.com/

[84] (2010) Red5. The Red5 Project. [Accessed: December 2010]. [Online]. Available: http://red5.org/

[85] (2010) The Apache HTTP Server Project. Apache Software Foundation. [Accessed: December 2010]. [Online]. Available: http://httpd.apache.org/

[86] K. S. Bhogal. (2004, June) Advanced JMS Messaging with OpenJMS. [Accessed: December 2010]. [Online]. Available: http://www.devx.com/Java/Article/21261/0/page/1

[87] (2010) HornetQ - Putting the Buzz in Messaging - JBoss Community. JBoss. [Accessed: December 2010]. [Online]. Available: http://www.jboss.org/hornetq

[88] T. Fox. (2010, February) JBoss HornetQ Sets Record SPECjms2007 Benchmark Results. [Accessed: December 2010]. [Online]. Available: http://www.timfox.me/2010/02/jboss-hornetq-sets-record-specjms2007.html

[89] C. Stasch. (2009, November) 52North SOS Data Modeling. [Accessed: December 2010]. [Online]. Available: https://wiki.52north.org/bin/view/Sensornet/SosDataModeling

[90] (2009, April) JFreeChart. Object Refinery Limited. [Accessed: December 2010]. [Online]. Available: http://www.jfree.org/jfreechart/

[91] (2010) Facebook Developers. Facebook Inc. [Accessed: December 2010]. [Online]. Available: http://developers.facebook.com/

[92] (2010, August) GPS API in S60 3rd Edition - Forum Nokia Wiki. Nokia Corporation. [Accessed: December 2010]. [Online]. Available: http://wiki.forum.nokia.com/index.php/GPS_API_in_S60_3rd_Edition