An Architecture for Socially Mobile Collaborative Sensing and its Implementation

by

Charles William Hawthorne Amick

S.B., Computer Science, Massachusetts Institute of Technology (2008)
S.B., Music, Massachusetts Institute of Technology (2008)

Submitted to the Department of Electrical Engineering and Computer Science
in Partial Fulfillment of the Requirements for the Degree of

Master of Engineering in Electrical Engineering and Computer Science

at the

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

June 2009

ARCHIVES

MASSACHUSETTS INSTITUTE OF TECHNOLOGY
JUL 2 0 2009
LIBRARIES

© Charles William Hawthorne Amick, MMIX. All rights reserved.

The author hereby grants to M.I.T. permission to reproduce and to distribute publicly paper and electronic copies of this thesis document in whole and in part in any medium now known or hereafter created.

Author ....
Department of Electrical Engineering and Computer Science
May 15, 2009

Certified by ....
David P. Reed
Adjunct Professor
MIT Media Lab
Thesis Supervisor

Accepted by ....
Arthur C. Smith
Professor of Electrical Engineering
Chairman, Department Committee on Graduate Theses

An Architecture for Socially Mobile Collaborative Sensing and its Implementation

by

Charles William Hawthorne Amick

Submitted to the Department of Electrical Engineering and Computer Science
on May 15, 2009, in partial fulfillment of the
requirements for the Degree of
Master of Engineering in Electrical Engineering and Computer Science

Abstract

We introduce the concept of "socially mobile collaborative sensing" (SMOCS), a collaborative process in which devices owned by different users gather and share contextual data and derive inferences. People are increasingly gathering contextual data (e.g. location [38]) for their own use and are beginning to share it through systems such as Twitter and Google Latitude. These systems are domain specific (they are designed for a single data type) and require all users to gather their context themselves; users must install software on a device capable of gathering rich contextual data (e.g. a smartphone). The SMOCS architecture enables users with less-capable devices to offload contextual sensing and reporting to physically proximate more-capable devices; in this way, SMOCS defines a truly viral architecture.

Using collaboration among devices that are found in the neighborhood of a mobile user to gather contextual data is now practical. The density of reachable devices in any particular person's neighborhood is growing with time. SMOCS is motivated by the need to structure such collaborations to support viral growth and evolution of applications that exploit such sensing.

In this thesis we define SMOCS and present an architecture and an implementation, called ContInt (Context Interleaving). The ContInt implementation is composed of two components: Ego, a distributed social network in which users maintain personally-owned "agents", and the ContInt plugin for Ego. The proposed SMOCS architecture enables the collection and distribution of contextual data and provides extensible interfaces to allow inference-deriving "plugins."

We evaluate ContInt in terms of a) scalability and performance, b) architectural extensibility and c) argue that its privacy model enables users to control their data. We conclude by proposing future work and our expectations of how SMOCS might evolve.

Thesis Supervisor: David P. Reed
Title: Adjunct Professor
MIT Media Lab

0.1 Acknowledgments

This thesis was made possible with the help of many people. In particular, I would like to thank:

- **David P. Reed**, my thesis advisor, for giving me invaluable feedback and allowing me the flexibility to explore technical areas of interest to me.
- **Andrew Lippman**, my co-advisor, for asking questions and helping me distill my ideas.

Andrew Lippman and David P. Reed took me on as a Research Assistant at the Media Lab. It has been an eye-opening experience that has not only been enjoyable, but has also been a fascinating education.

- **Kwan Hong Lee** for his support, especially in my senior year. Kwan vouched for me as I attempted to join the Viral Communications group to pursue this thesis.
- **Polychronis Ypodimatopoulos** with whom many of the ideas in this thesis were implemented. Pol helped me to keep working when things were going badly, and we had a great time when things were going well.
- **Nadav Aharony, David Gauthier, Inna Koyrakh, Fulu Li** and **Dawei Shen**; together with Kwan and Pol, the Viral Communications group. They made me feel welcome from the outset and provided stimulation.
- **John and Elizabeth O'Beirne Ranelagh** and **Rachel Godshaw** for their help editing this thesis and their feedback from a non-computer science point of view.
- **My friends and family** for putting up with the ups and downs of the thesis process.

Contents

0.1 Acknowledgments 3

1 Introduction 11

2 Socially Mobile Collaborative Sensing 17

2.1 Definition: Collaborative Sensing 17

2.2 Definition: Socially Mobile Collaborative Sensing 19

2.3 Motivation 21

2.3.1 Personal sensing systems 21

2.3.2 Collaborative sensing systems 22

2.4 Foundations 22

2.4.1 Ubiquitous computing 22

2.4.2 Participatory sensing 24

2.4.3 Physical presence detection 24

2.4.4 Information accountability 25

2.4.5 Personal sensing 26

2.4.6 Personal collective intelligence 26

3 Architecture 27

3.1 Overview 27

3.2 Design principles 28

3.3 Architectural elements 29

3.3.1 Entity 29

3.3.2 Agent 30

3.3.3 Relationship 30

3.3.4 Aggregator 31

3.3.5 Identifier 31
3.3.6 Sensor 32
3.3.7 Device 32
3.4 Architectural schematics 33
3.5 Architectural subsystems 34
3.5.1 Distributed eventing 35
3.5.2 Network interoperability 35
3.5.3 Plugin framework 36
3.5.4 Privacy controls 37
3.6 Protocols and API 37
3.6.1 Protocols 37
3.6.2 API 39
3.7 Object models 40
3.7.1 Agent 40
3.7.2 User 40
3.7.3 Subscription 42
3.7.4 Aggregator 42
3.7.5 Device 42
3.7.6 FixedMeta 43
3.7.7 DeviceOption 43
3.7.8 Sensor 43
3.7.9 Identifier 44
3.7.10 Sensing 44
3.7.11 SensingMetadata 44
3.7.12 Report 45
3.7.13 ReportMetadata 45
3.8 Implementation 46

4 Evaluation **47**
4.1 Scalability and performance 47
4.1.1 Metrics 47
4.1.2 Scenarios 48

4.1.3 Results: Single agent 50
4.1.4 Results: Multi-agent 53
4.1.5 Discussion 56
4.2 Architectural extensibility 60
4.2.1 LabTracker 60
4.2.2 Open Spaces 61
4.2.3 TinyBT 63
4.2.4 RemoteSense 63
4.2.5 Advanced privacy controls 64
4.3 Privacy 65

5 Conclusion **67**
5.1 Overview 67
5.2 Future work 69

A Implementation **71**
A.1 Implementation decisions 71
A.2 RESTful design patterns 74
A.3 Inter-Agent security protocols 75
A.4 Naming conventions 78
A.5 API generation 80
A.6 Web-based user interface 82
A.7 Distributed eventing 87
A.8 Plugin framework 88
A.9 Tentacle implementation 89
A.9.1 Workflow 90
A.10 Source code 93

List of Figures

2-1 Collaborative sensing. Alice's device advertises its sensing capabilities. Neighboring devices can connect to her device and query its GPS sensor for their location. 18
2-2 Socially mobile collaborative sensing. Bob has shared his MAC address with Alice. Alice's device detects the presence of Bob's device in its contextual environment, and reports its contextual data to Bob's agent over the network. 20

3-1 High-level conceptual diagram showing the SMOCS architecture. 28
3-2 Directionality of relationships. Sources provide data, and destinations receive data. 31
3-3 Workflow of raw contextual data being transformed into reports and shared with other agents. 33
3-4 Interaction between agents and aggregators through relationships. 34
3-5 Figure detailing the relationship between agents, identifiers, sensors and devices. 35
3-6 Distributed eventing architecture. Alice subscribes to Bob's "context changed" event and is notified when the event is raised. 36

4-1 The ONE simulator. Simulated people are seen. Each person carries a simulated device that periodically reports location, Bluetooth neighbors and temperature to the ContInt plugin running on an Ego agent. 49
4-2 Graph showing size of the agent's database over time in the single agent scenario. 50
4-3 Graph showing network usage of all agents and devices in the system in the single agent scenario. 51

4-4 Graph showing processor usage of machine hosting single agent scenario. . 52
4-5 Graph showing memory usage of apache2 instances on machine hosting single agent scenario. 52
4-6 Graph showing aggregate size of all agent databases over time in the multi-agent scenario. 54
4-7 Graph showing network usage of all agents and devices in the system in the multi-agent scenario. 55
4-8 Graph showing processor usage of machine hosting multi-agent scenario. . 55
4-9 Graph showing memory usage of apache2 instances on machine hosting multi-agent scenario. 56
4-10 LabTracker user interface, showing when members of the Viral Communications group were last in the lab area. 60
4-11 Open Spaces being used by two people carrying Amulets. 61
4-12 Open Spaces demonstration showing the calendars of two users side by side at the screen. The presence of the users was detected with ContInt, and their calendars were retrieved from their Ego agents. 62
4-13 RemoteSense application showing a stream of contextual data from sensors run on devices of a remote agent. 64

A-1 Agent IA function encryption and authentication workflow. 79
A-2 User interface of the main screen of an agent. 82
A-3 User interface showing a list of all friends of the agent's owner. 83
A-4 User interface showing timeline of recent status updates from the friends of the agent's owner. 83
A-5 User interface showing device management screen in the ContInt Ego plugin. 84
A-6 User interface showing identifier management screen in the ContInt Ego plugin. 84
A-7 User interface showing geographic region management in the ContInt Ego plugin. 85
A-8 User interface showing latest contextual data in the ContInt Ego plugin. . . 86
A-9 User interface showing plugin management in Ego. 89

Chapter 1

Introduction

The diverse electronic devices we use daily are increasingly being put to the task of sensing the physical world around us. Modern technologies enable mobile devices, such as cellular telephones, to sense acceleration, to calculate latitude and longitude to within 10 meters, and to scan multiple radio networks (802.11, Bluetooth) for device and service discovery. Statically-deployed infrastructure devices are even more powerful - they draw electricity from the grid, need not be portable, and are wired into reliable networks; they can support many power-hungry sensors (e.g. video, audio.) The output of these sensors is contextual data: it describes the context in which the sensor operates.

Both mobile and infrastructure devices are becoming more prevalent and more powerful. It is probable that mobile devices will become increasingly powerful and thus support sensors capable of gathering more contextual data; the growth of smartphone market share supports this [52]. In parallel, infrastructure-hosted sensors will increase in prevalence: the British government has already deployed 4.5 million CCTV cameras in the UK with more planned [39], and research groups at the MIT Media Lab have wired the entire lab with sensors capable of recording audio, video and Bluetooth device presence. The pervasiveness of mobile and infrastructure devices and their sensors suggests the potential for context-driven applications that can scale.

Context-driven applications are applications that perform actions in response to changes in a person's contextual environment. An example application is one that emails a reminder to complete a task on a person's to-do list as soon as the person comes into range of a place where that task can be accomplished. Not all applications need react in real-

time; some can mine historical contextual data to draw inferences. An example of such an application is one that learns the social interaction patterns of you and your friends. The application could determine if you are neglecting a friend and alert you. Another class of applications are community-wide applications: users share subsets of gathered context with community aggregators in order to act as a grassroots, participatory sensing platform [5]. For instance, users could share their current health with the local government in order to track flu outbreaks.

More and more individuals are gathering contextual data in order to use such applications. Currently, location information is the primary form of contextual data gathered because of the vast potential seen for location-aware applications. Yahoo!'s Fire Eagle [31] allows users to run software on their cell phone that gathers location data and reports it back to their Fire Eagle account. Google's Latitude [23] software performs a similar task. Both systems enable users to share their location with their friends and publicize it to the world.

Twitter [58] takes sharing one step further, by allowing users to post short (140 character) status updates containing arbitrary data. Users can subscribe to their friend's "tweets." This is a simple publish-subscribe model: users publish ("tweet") to their subscribers ("followers"). An inadvertent byproduct of this architecture is its possible use as a centralized announcement mechanism for any information that can be encoded into a 140 character string. Twitter is increasingly used to publish rich contextual information (such as latitude, longitude, images of surroundings and mood). This publicly accessible tweet information is mined by third-party applications and used in "mashups" - generally, an application formed by the cross-pollination of two distinct datasets. Examples of popular mashups include Twopular.com [12], a website that aggregates current trends on Twitter based upon what people are tweeting, and Portwiture.com [55] which chooses photographs from Flickr [32] that match the content of posted status updates. The wide variety of mashups and the simplicity of the system has driven Twitter's success and fuelled its rapid growth to over 5 million users [45] in under three years. Mashups are particularly interesting: they hint at the possibility of community-wide aggregation and data mining systems.

Twitter, Fire Eagle and Latitude serve as prototypical demonstrations of people's interest in gathering contextual data and sharing it. However, they fail to meet the requirements of a context-gathering system that could have massive adoption:

1. **Privacy** - Privacy is handled by all of these systems, but in an arcane manner. The approach is "all-or-nothing" - data describing a user is either available only to that user or all of it is available to specified peers. This has worked so far because each person has their own view of what privacy should entail. Some take pride in publicizing images of themselves engaging in crude behavior; others are conscious of the consequences ([42] and [7] are examples of employees being fired for their behavior outside work) and restrict sharing to a subset of trusted peers. The privacy models of Twitter, Fire Eagle and Latitude are designed with these desires in mind and apparently satisfy users. However, this all-or-nothing approach means that as soon as data is placed on to the network it can never be removed, leaving users out of control of their data. Our privacy model puts control first: if users feel in control of not only who can access their data but how it is stored, they will be likely to trust the system more. Control is increasingly important as more real-time context-driven applications are developed; such systems could be compromised and their data used to cause harm. Stalking is a legitimate fear in location-aware applications.

2. **Exploit virality** - A viral system is one that can grow and evolve quickly in a market oriented economy and a decentralized democratic culture [37]. Some properties of viral systems include:

   (a) A low barrier to entry.

   (b) The utility of the system growing proportionally with the number of users of the system.

   (c) Utility from the outset: otherwise, attracting users to the system will be impossible.

   Points b) and c) are generally true of existing social networking websites (e.g. Facebook, MySpace) in which users form friendships, send each other messages, post images, etc. Point a) is not true of the existing services in terms of context gathering: users have to run specialized software on their mobile device in order to gather and report contextual data.

We argue that existing architectures for contextual data gathering and sharing do not exploit virality, nor do they offer appropriate privacy models. This is no surprise, as the

context-gathering components of Twitter have been added post-inception, and Latitude and Fire Eagle are location-centric and fueled by the desire to collect user locations for corporate data-mining purposes. This leads us to define an architecture that does satisfy the two requirements.

Our architecture is based on our concept of "socially mobile collaborative sensing" (SMOCS). SMOCS defines protocols for gathering, storing and sharing of contextual data. In particular, SMOCS defines how a mobile device can interact with others in its neighborhood in order to gather richer contextual data than it could on its own. This means that users with less-capable devices are able to gather data that they otherwise could not have; a user without GPS could approximate their location through a nearby device with a GPS sensor. This architecture leads to a truly viral system as many users have the opportunity to get involved.

We use the idea of "collaborative sensing" (proximate devices sharing sensing capabilities) to derive SMOCS. The "social" aspect of SMOCS is introduced by our requirement that the system be as viral as possible: we assume some devices in the system cannot run any specialized software whatsoever. Instead, users run a "dumb" device that can merely broadcast a presence identifier. Identifiers are shared through relationships formed between users of the system. We present the mechanism of SMOCS in the next chapter.

The future of contextual data gathering and sharing is a system that is both privacy-centric and truly viral. The SMOCS architecture satisfies these requirements: privacy is ensured by strict access controls and data ownership, while virality is exploited by allowing offloading of sensing to trusted peers.

In this thesis we define SMOCS, derive the SMOCS architecture and present our implementation of the architecture, ContInt (Context Interleaving). ContInt enables users to gather contextual data with the sensors in their personal device, store this data in a privacy-centric manner, and subsequently share subsets of data with trusted users through high granularity access-control mechanisms. We evaluate ContInt's scalability and performance properties, its architectural extensibility and argue that its privacy model is adequate and puts the user in control of their own data.

The key contributions of this thesis are the definition of SMOCS, the SMOCS architecture and its implementation, ContInt. The thesis is organized as follows: in chapter 2 we present the motivation behind SMOCS and its definition, as well as an overview of

related work; in chapter 3 we discuss the architecture of SMOCS, and refer the reader to appropriate appendices for implementation details; chapter 4 evaluates our implementation; chapter 5 concludes with a discussion of future work in this area, and how ContInt could complement and enable this work.

Chapter 2

Socially Mobile Collaborative Sensing

In this chapter we discuss the term "socially mobile collaborative sensing" (SMOCS). We begin by defining collaborative sensing and use its shortcomings to derive and motivate our definition of SMOCS. We then present the motivation behind the concept and show cases where a SMOCS architecture would be useful. Finally, related work, particularly in the field of ubiquitous computing and participatory sensing, is discussed.

2.1 Definition: Collaborative Sensing

In collaborative sensing, devices with distinct sensing capabilities collaborate to provide each other with rich contextual data that each device alone could not otherwise sense. For example, if Alice's device has a GPS receiver it advertises this capability over Bluetooth's Service Discovery Protocol (SDP) [53]. Bluetooth is range-limited, with the most common (class 2) transmitters reaching ranges of up to 10m [60], so only neighboring devices will hear Alice's advertisement and be able to connect to her device to query its GPS receiver for coordinates. Figure 2-1 illustrates this example: Bob's less-capable device can query Alice's more-capable device for contextual data.

This concept works especially well for contextual data, because two physically proximate devices are likely to exist in a similar context. If Bob is within a few meters of Alice, Bob's less-capable device can outsource sensing to Alice's more-capable device. Bluetooth

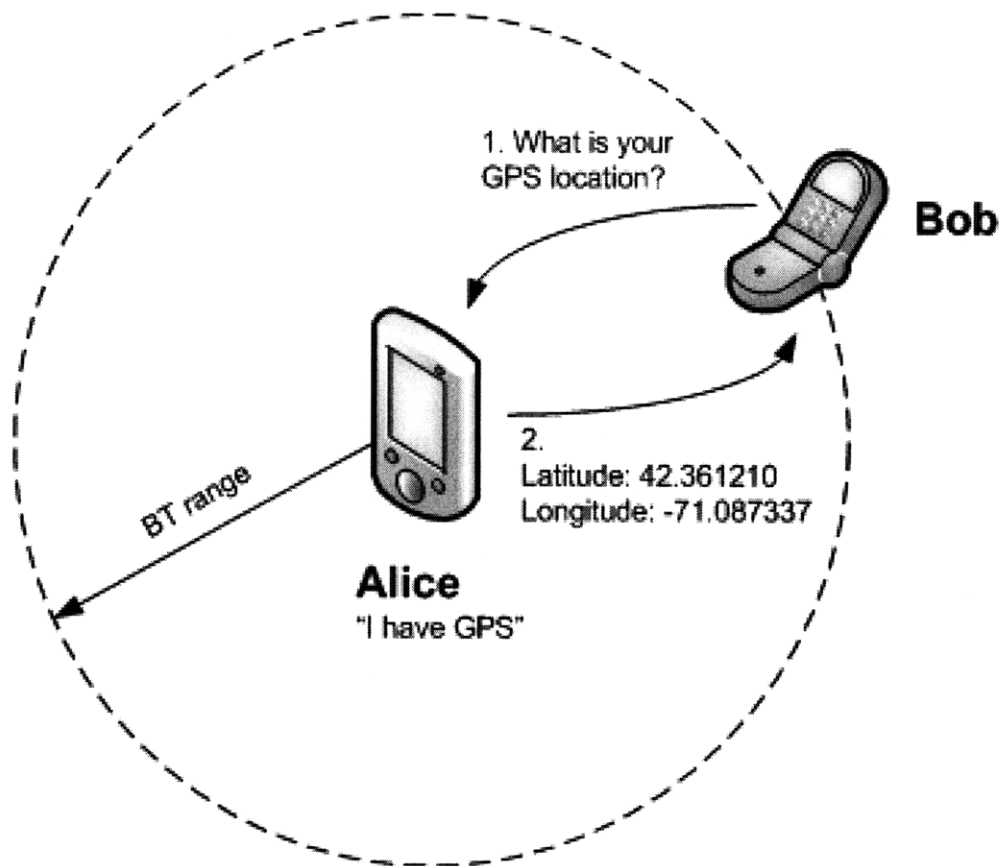

Figure 2-1: Collaborative sensing. Alice's device advertises its sensing capabilities. Neighboring devices can connect to her device and query its GPS sensor for their location.

is used for hyper-local peer and service discovery; its short-range nature enables collaborative sensing.

Collaborative sensing is presented under the assumption that all users in the system must have smart devices that are programmable and can communicate with each other in order to share sensing capabilities. Though cellular telephones (the most prevalent mobile sensing platform) are becoming increasingly heterogeneous in capability (smarter, not dumber), we assert from past experience that there will always be devices of varying capabilities. Historically, this has been due either to network operators willing to cripple device capabilities (such as Verizon's move to disable file sharing over Bluetooth in the Motorola RAZR V3 [36]) or device manufacturers failing to provide openly programmable frameworks. The iPhone is an excellent example of a crippled programming platform: the device itself has Bluetooth, 802.11, GPS and WPS (WiFi positioning provided by Skyhook [54]). Unfortunately, no Bluetooth API is provided, so Bluetooth applications cannot be de-

veloped using the standard development frameworks. Because of competing systems and proprietary hardware and software, the capabilities and programmability of smart devices cannot be relied upon if we wish to create a truly viral architecture.

2.2 Definition: Socially Mobile Collaborative Sensing

SMOCS improves upon collaborative sensing by removing the requirement that devices be smart and programmable. There are two requirements of collaborative sensing that must be removed in order for "dumb" devices to take part in the system:

1. **Gathering** - Dumb devices cannot gather contextual data, so they must outsource this gathering to smart devices.

2. **Reporting** - We assume that dumb devices cannot actively communicate with the outside world, so they cannot report their context to any external web services (e.g. Yahoo Fire Eagle), nor can they proactively ask smart devices around them to perform this task for them.

SMOCS removes these requirements with a simple observation: most dumb devices have Bluetooth chips and can be discoverable; that is, they can broadcast their globally unique MAC address. If a person with a dumb device is willing to make their device discoverable, they can share their device's identifying MAC with other trusted smart devices. The smart devices gather context on behalf of their user anyway; if they identify a known MAC address they can automatically share their gathered context with the identified user. As we argued when describing collaborative sensing, this gathered context is a good approximation of the actual context of the user, as the detecting and detected devices are necessarily within approximately 10 meters of each other.

Note that, since dumb devices cannot have context reported directly to them from smart devices, an extra piece of infrastructure is required for smart devices to report contextual data to. We call this infrastructure an "agent". An agent serves as the context-reporting endpoint for a particular user. There is a one-to-one relationship between users and agents: an agent represents a user in SMOCS. We assume that agents are hosted somewhere on the Internet and, for simplicity, are always connected. Agents are identified by URLs. Further details are given in the next chapter.

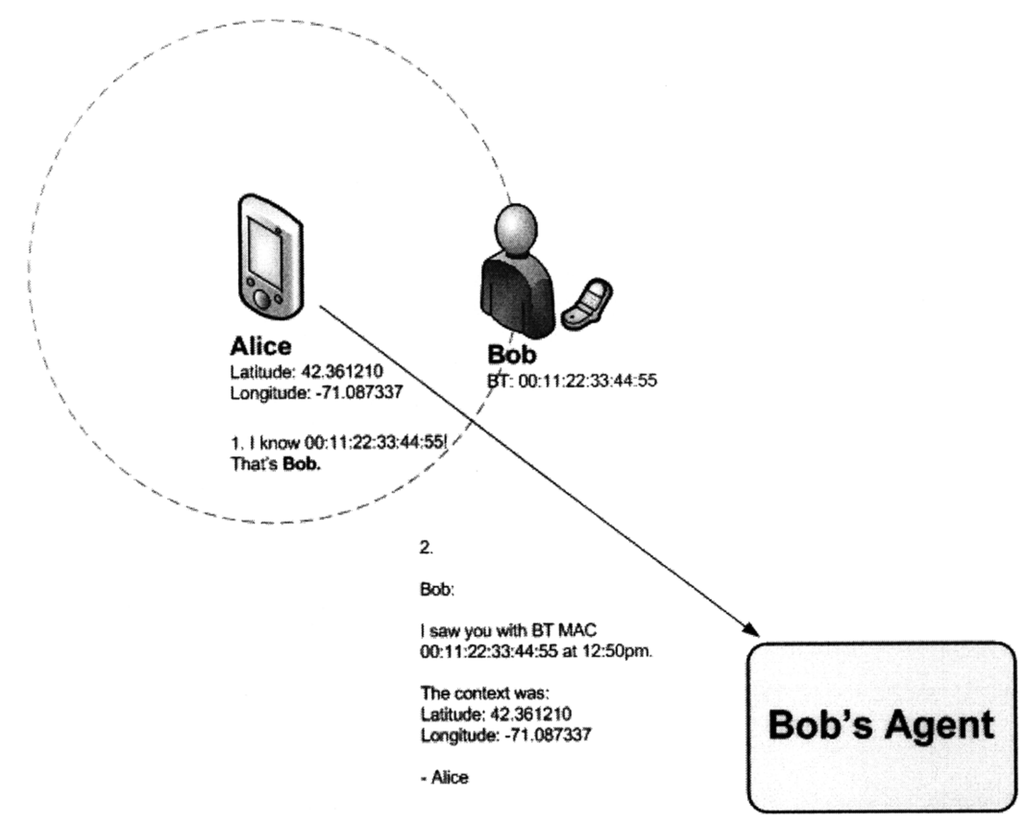

Figure 2-2: Socially mobile collaborative sensing. Bob has shared his MAC address with Alice. Alice's device detects the presence of Bob's device in its contextual environment, and reports its contextual data to Bob's agent over the network.

Consider again Alice and Bob, but now in the example of SMOCS. Alice has a smart device with a number of sensors (temperature, location, audio, video). Bob has a dumb device. Bob puts his device in Bluetooth discoverable mode and shares his device's MAC address and the URL of his agent with Alice. When Alice's device detects Bob's device, Alice's device infers that Bob is physically present. Alice's device sends the gathered contextual data to Bob's agent. Figure 2-2 illustrates this example.

To participate in a SMOCS system, users with dumb devices must share personally identifying data (their personal device's MAC address) with other users who own smart devices. The smart devices gather contextual data on behalf of the dumb devices and send it to the agent of the device's owner. Because the MAC address of a device does not typically change over its lifetime, it is likely that a user will share their MAC only with people they trust - that is, people with whom they have a pre-established social relationship. This is where the "social" in SMOCS comes from: we assume that only close friends or other-

wise trusted entities (people, organizations) will be given this privileged data.

Note that an inference is made when Alice detects Bob's device that Bob is physically present. We assume that users of a SMOCS system only share identifiers of devices they are likely to be carrying. It is possible to dictate to one's agent the current device being carried so as to filter out contextual reports of devices not applicable to oneself.

2.3 Motivation

The motivation for SMOCS can be distilled into two separate sets of motivation: 1) the motivation for personal sensing systems in general and 2) the motivation for collaborating with others to enrich each other's gathered contextual data.

2.3.1 Personal sensing systems

There are a number of motivations for the creation and use of personal sensing systems:

1. **Personal intelligence** - We are increasingly able to gather data about our lives that can be used for personal understanding. For example, if Alice comes down with a serious illness her past contextual patterns can be mined to determine what might have caused it. This could help researchers understand the the impact of patterns of exposure to certain environments. Another example would be correlation between productivity and location: Bob could make a judgment about whether working in a library enabled greater productivity.

2. **Group intelligence** - Personal data can be shared amongst a group of users to enable group intelligence applications. For example, an application could aggregate location information from users in a community to create a map displaying which parts of a city are busy at different times of a day. Friends could share their historical shopping bills to compare how much they spend on groceries. Group-wide applications can target groups on different scales, from the global scale to the community scale and down to the social network scale.

3. **Ubiquitous computing-style interactions** - Users of a SMOCS system could share their real-time location information with an application that allowed them to interact with digitized resources in the world around them; for instance, one could project

personal data on a publicly available computer screen by virtue of being near it if the screen has been allowed to access that data.

2.3.2 Collaborative sensing systems

Collaborative sensing systems are motivated by the observation that many devices with diverse capabilities are reachable via short-range radios at any given time; the number is only expected to increase. If two smart devices can communicate, they should be allowed to collaborate in order to gather rich contextual data.

The motivation for SMOCS is to remove from collaborative sensing the requirement that all devices be smart. We remove this requirement so as to build a viral architecture capable of supporting many users with devices of diverse capabilities. SMOCS enables users with dumb devices to collect contextual data in return for sharing identifying information with trusted peers. Smart devices can also benefit: sensing capabilities are heterogeneous across devices, so two smart devices could have different capabilities. By working together, devices can gather richer contextual data.

Of course, there is an inherent trade-off between identity privacy and contextual data richness. If a user chooses to share their identifiers with everyone they will receive high quality collaboratively sensed context. Conversely, if a user with a dumb device chooses not to share their identifiers with smart devices or chooses not to leave their device in discoverable mode, the user will collect no sensed contextual data.

2.4 Foundations

In this section we overview the foundations upon which SMOCS is built. Some are merely related, others are built upon directly. We present several fields of research that investigate similar concepts and discuss some solutions that exist in those fields.

2.4.1 Ubiquitous computing

Ubiquitous computing is a model of human-computer interaction that assumes integration of information processing into everyday objects. Passive objects, such as TVs, are transformed into active objects that a passerby can use. For instance, a computer-equipped TV might be able to sense the presence of a passerby and display content from the passerby's

device on the TV screen. Most ubiquitous computing systems require digitization of contextual information - in particular, physical presence - in order for interactions to occur.

Privacy is a great concern in ubiquitous computing applications. Users must advertise their identity in a way which allows them to interact with their digitized environment whilst maintaining privacy. Many different architectural solutions have been proposed to deal with the problem of securely identifying oneself to resources in one's surroundings [28, 64, 41, 48, 51, 35]. SMOCS works equally well if identifiers are secure or unsecure, so adopting one of the above solutions would be possible, but would currently require extra software running on personal devices. We want the system to be as viral as possible, so our current implementation simply relies upon unsecure Bluetooth MAC addresses to serve as personal identifiers.

A novel system design in this field is Confab[28]. In Confab, users report their context information to a personal "infospace." The infospace is akin to our concept of an "agent". Infospaces manage context for a user and broker queries from other users. Confab assumes context is gathered by, and reported from, the personal device of a user to their personal infospace; thus privacy is maintained.

Confab does not allow a situation that is key for creating a viral context gathering system: users must report their own sensed context. They cannot delegate this task to trusted third parties, such as their close friends or sensor networks run by a trusted institution or group. There is also limited opportunity in Confab to use historical context in order to participate in third party applications. Furthermore, group-wide intelligence applications are not part of the architecture.

PlaceLab [51, 35] is a marginally related project, in which devices localize themselves using radio beacons from known coordinates. This relates to our work in that collaboration that takes place to localize oneself by using intelligent devices in one's surroundings. Location is just one type of contextual data that SMOCS gathers. We can learn from PlaceLab's results if we are interested in deploying fine-grained localization indoors, but we can also assume that location can be determined increasingly easily using WiFi localization techniques such as those provided by SkyHook [54].

2.4.2 Participatory sensing

Participatory sensing is a relatively new field related to sensor networks. A participatory sensor network is one in which users collaborate in order to sense their environment. In [5], Burke et al. note that the increasing abilities and ubiquity of cellular telephones makes the platform perfect for participatory sensing networks. They give numerous examples, one of which is public health; a population voluntarily providing current medical symptoms through a small application running on their cell phone could allow authorities to rapidly identify and handle environment-specific health issues.

Another interesting example of participatory sensing is "hitchhiking," as described in [57]. In hitchhiking, users report to a central server the number of other people they detect in certain locations. The reports are anonymized so the exact locations of reporters are not disclosed. Applications built on top of this concept include one to determine seating space in local coffee shops.

Participatory sensing is a foundation of our work. As noted, SMOCS enables sharing of personally gathered contextual data, thus conceptually allowing aggregation across a number of users. SMOCS could be used as a participatory sensing platform.

2.4.3 Physical presence detection

The virality of a SMOCS architecture depends upon users with less capable devices being able to share personally identifying pieces of data with others so that their physical presence can be detected and contextual data shared with them. One of the fundamental challenges is to determine the presence of neighboring devices and the identities of their owners.

Presence is a crucial problem to solve in the world of ad-hoc distributed systems. These systems have no centralized control; devices making up the system must be able to discover each other's presence so that they can collaborate to form a network. Many distributed systems have their own protocols for presence because each has slightly different requirements. Some protocols build hyper-local networks between one-hop neighbors. Others create multi-hop networks that span ranges greater than any single device's radio range alone.

In our research group, we have developed three separate protocols solving three slightly

different presence-related problems. The most widely used of these is Cerebro [63], a lightweight, scalable protocol that enables devices in the same network subnet to discover each other's physical presence.

Cerebro is a relatively mature protocol in use by the One Laptop Per Child (OLPC) [8] initiative. It provides local-area collaborative applications with the digital presence required to operate. Cerebro was designed for efficient device discovery in ad-hoc networks but can run on infrastructure-based networks too, with the caveat that only devices in the same subnet can discover each other.

Cerebro works by having all devices emit beacons. Beacons are small packets containing the ID of the device emitting the beacon and the IDs of all of the other devices the emitter senses. Over time, all reachable devices running the protocol learn of each other. Once every device knows of each other, devices can send point-to-point queries for more detailed identity information.

Cerebro is too advanced for our needs. SMOCS can use it, but would under-utilize it; SMOCS merely needs information on 1-hop neighbors. The same information can be gathered using Bluetooth device scans, with the added advantage that most cell phones can enter Bluetooth discoverable mode whereas most cannot run Cerebro.

2.4.4 Information accountability

Information accountability, a term coined by Weitzner et al. in [59], notes the emerging realization that existing information usage policies are flawed. In traditional systems, data security is maintained by strict access control policies. Once compromised, private data may be made freely available to everyone. (Consider the Windows 2000 source code leak [10].) Weitzner et al. argue that this "hide it or lose it" approach is wrong, and that technology should be augmented to support information accountability and appropriate use, rather than information security and access restriction. The idea is that owners of data will be able to know when the data is viewed and determine whether such viewing was appropriate; if not, they will be able to hold the viewer accountable.

This concept should be (but is currently not) applied readily to a SMOCS-implementing system; in SMOCS there is a key privacy hole: the sharing of identifiers. Once an identifier is shared with a friend, there is nothing stopping that "friend" from giving away that identifier for whatever reason (personal gain, retribution, etc.) Thus, it is important that

users be able to track the movement of their personal data through the network as much as possible. We have not implemented this functionality in ContInt, but note that it can be developed.

2.4.5 Personal sensing

Personal Sensing is the body of work related to gathering personal contextual data and building applications that use it. Personal Sensing systems have been mostly built around the cell phone as it is indeed a sensing platform that tends to be carried by its owner and can be used to gather personal context.

CenceMe [41] is a concrete implementation of a cell phone based sensing system. CenceMe aims to infer high-level contextual information based on raw senor data gathered by phone-based sensors. Using learning algorithms, CenceMe's authors show that it is possible to infer activity, location, conversation state, etc. CenceMe has been released as an application for the iPhone.

CenceMe might appear to do many of the things we propose to do, but it does not. CenceMe is entirely about human context and has no interest in the architectural challenges of designing for ubiquitous computing environments. Furthermore, CenceMe does not allow users to collaborate to gather contextual data. In [41], the authors ask users without accelerometers in their devices to carry around a custom built Bluetooth accelerometer. This is a hint that a design decision should be made; asking users to carry extra peripherals to augment their sensing capabilities does not scale.

2.4.6 Personal collective intelligence

Personal Collective Intelligence is an idea that has been fostered in parallel to our work on SMOCS. In SMOCS, we examine the possibility of collecting personal data and then sharing this information to create a group-wide intelligence dataset. Personal Collective Intelligence (PCI) is essentially the same idea. (If anything, SMOCS is a PCI-compatible architecture.) The work is currently in early stages of research by Dr. Luis Sarmenta in the MIT Media Lab. At the time of writing of this thesis, collaboration between PCI and the work that is presented in this thesis is in its early stages.

Chapter 3

Architecture

In this chapter we present our proposed SMOCS architecture. Beginning with a high-level overview, we describe the requirements, our design principles and how they led to the architecture. We then define the elements that make up the architecture and conclude with a schematic figure. Where necessary, implementation details are set out in appendix A.

3.1 Overview

The primary goal of the SMOCS architecture is to enable users with smart devices to run software on these devices that can gather and report contextual data back to the user's agent. Users with dumb devices, or those with smart devices wanting richer contextual data, can delegate the task of gathering and reporting data to the smart devices of other trusted users. These smart devices can be mobile devices or infrastructure (static) devices. A smart device reports to the agent of a dumb device if the smart device detects the dumb device and knows how to reach the agent of the owner of the dumb device. Figure 2-2 showed this workflow.

The SMOCS architecture defines a distributed network of user-controlled "agents" that interact with each other in order to share contextual data, and gather it on each other's behalf. The resulting network resembles a peer-to-peer file sharing network, as shown in Figure 3-1.

In the following sections we justify the architecture of the system and detail the responsibilities of agents, the meaning of relationships, and how contextual data can be gathered and shared between agents.

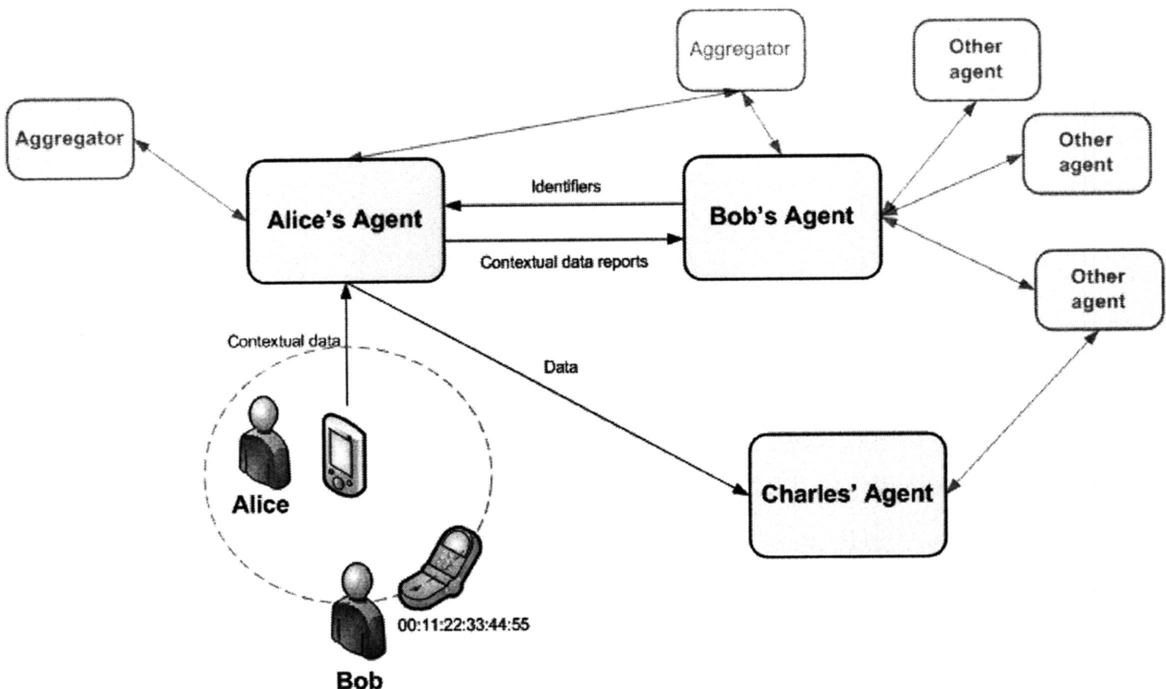

Figure 3-1: High-level conceptual diagram showing the SMOCS architecture.

3.2 Design principles

The following design principles were adhered to when developing the system architecture:

- **Privacy** - The contextual data gathered can be private and sensitive. Some users might not want this data stored in servers owned by others for fear of it leaking or being misused, so it must be possible for users to run their own agents for the storage, retrieval and mining of gathered contextual data. For those uninterested in running an agent themselves, it must be possible to delegate hosting of an agent to a third-party service (much like one would delegate hosting of a website.)

- **Contextual extensibility** - The system must allow future, unforeseen sources of context to be gathered, reported and shared.

- **Architectural extensibility** - The system must provide clear, robust APIs that can support development of future context-driven applications. The system should allow developers to build "pluggable" components rapidly.

- **Virality** - Most importantly, the system must exploit virality. (If it did not, it would

not be a SMOCS system.) It must have a minimal barrier to entry to encourage adoption, and it must add value as more users adopt it so as to be constantly attractive.

These principles lead to the following requirements:

- **Agent locations** - Agents could theoretically be built into the devices people carry; however, this would defeat the system since it would no longer be viral, as personal devices would have to be able to run agents. Since smart devices must report context to the agents of dumb devices, we assume that agents must be reachable by a network. The obvious network of choice is the Internet; thus, agents should be reachable by a URL on the Internet.
- **Network structure** - Agents need to communicate with each other in order to share device identifiers, report contextual data and (if the agent's owner desires) share contextual data with friends. Furthermore, privacy requires that it be possible for a user to run their own agent, thus the system cannot be centralized. Given these requirements, we immediately see that agents must be implemented so as to form a distributed network amongst themselves.

These requirements dictate a certain amount of the SMOCS architecture. The resulting architecture must take the form of a distributed system, and the network powering the system should be the Internet.

3.3 Architectural elements

In this section we describe the abstract architectural elements that make up the SMOCS architecture.

3.3.1 Entity

All of the examples we have presented to this point have involved human users interacting with SMOCS systems. We have stated that agents represent a human user, and that humans own the devices that gather and report contextual data. We now expand our definition of a user to include inanimate entities: a building, for example, could be represented by an agent and have a set of devices with sensors under its control. People, buildings and other inanimate entities are referred to with the umbrella term "entities" for simplicity.

3.3.2 Agent

An agent is the representative of an entity in the SMOCS network. An agent could represent a person called Alice; it could also represent an inanimate object, such as a public TV screen or a more abstract entity such as a group (e.g. a sports team could have an agent). Alice stores personal data in her agent (e.g. her to-do lists, historical location data and contact details) and this data is provided to context-driven applications that she chooses to install and run on her agent. Alice's agent can communicate with other agents and vice versa by forming agreed relationships with others; contextual data can be reported to her agent and shared with the agents of her friends.

Arbitrary data can be stored in an agent. The agent maintains all data stored in it, the relationships that it is participating in and the access rules to the data that it stores. Agents can proactively perform tasks, such as maintenance on the dataset of the user they represent, or mining of that dataset. Agents can be interacted with only by their owner: Bob can only view Alice's exposed data through the interface of his own agent.

3.3.3 Relationship

A relationship is formed between two agents when the owners of the agents choose to share data with each other. A relationship can represent a real-life relationship between two humans, or a more abstract notion of a "relationship" between a human and a static entity. For instance, Bob might form a relationship with a sports team if he wished to be notified when the team has extra tickets for sale for an upcoming event. This is not a relationship in the sense of friendship, but rather an agreement with the team that states that Bob should be alerted of changes to information. For simplicity we refer to two agents in a relationship as "friends."

Relationships are directional and have a source and a destination. The direction represents the flow of data: if Alice becomes friends with Bob, Alice is "opening up" to Bob, so Alice is a "source" of data to Bob (the "destination" of the information.) Figure 3-2 demonstrates this concept.

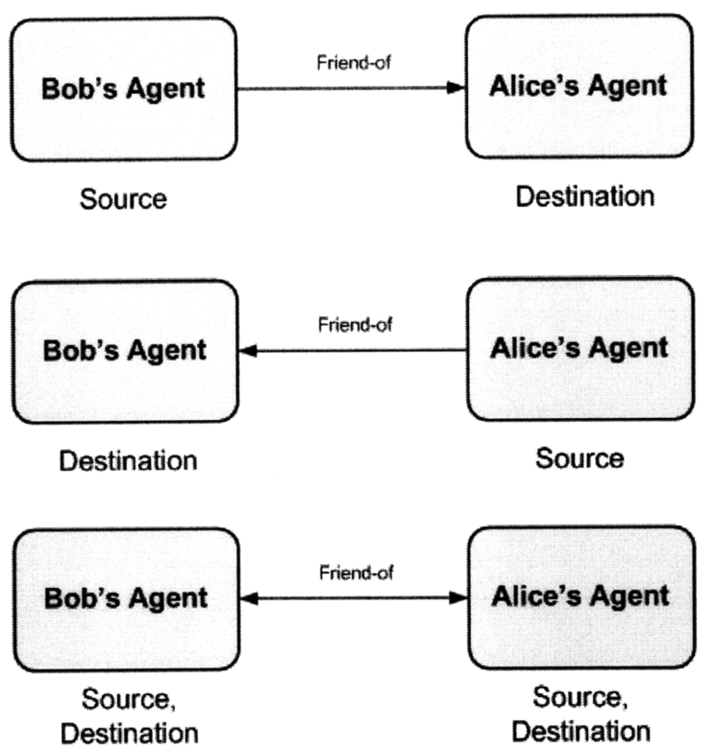

Figure 3-2: Directionality of relationships. Sources provide data, and destinations receive data.

3.3.4 Aggregator

An aggregator is a service that periodically collects data from a set of participating agents and mines this data to extract higher-level meaning than could be gathered from the data set of any single agent. Aggregators enable group-wide intelligence. An aggregator performs the same core functionality as an agent, and could be implemented as an extension to an agent; it is introduced as an architectural element because of its conceptual importance. We discuss its implementation in appendix A.

An entity makes their agent participate in an aggregator if the aggregator provides a valuable service. This value could be a "cool" factor (e.g. the aggregator creates interesting visualizations of a dataset) or could be actual value (e.g. the aggregator provides some group-wide intelligence that could otherwise not be gained.)

3.3.5 Identifier

An identifier is a piece of data that, when sensed, indicates that an entity is physically present. For instance, the MAC address of a Bluetooth device is globally unique and can

be sensed with a Bluetooth radio. If a MAC is sensed and the owner of the MAC address is known, it can be determined that a device of the owner is physically present. If the device is known to be carried by the entity, it can be inferred that the entity itself is present. Common identifiers include Bluetooth MAC addresses, a set of photographs used for facial recognition, or a unique ID in an ad-hoc distributed network such as Cerebro [63]. Identifiers are generally tuples of the form *<identification pattern, network, owning agent identity>*. For instance, <AA:BB:CC:DD:EE:FF, Bluetooth, Alice's agent>.

3.3.6 Sensor

A sensor is an electronic device that can measure a physical quantity within its environment and report this quantity as a digital value. The output of a sensor is contextual data that describes the contextual environment of the sensor. Typical sensors include Bluetooth and 802.11 radios, GPS receivers, temperature sensors, cameras and microphones. These sensors gather different types of contextual data: for instance, Bluetooth can report the MAC addresses of all discoverable devices within its environment; GPS receivers can report high-granularity location information.

All sensors can gather contextual data but only a subset of sensors can identify entities from this data. Bluetooth devices can detect MAC address identifiers on the Bluetooth network; if the sensing device has an identifier for the discovered MAC, an identification can be made. GPS sensors and temperature sensors cannot make identifications. In general, short range radios, video and audio sensors can make identifications.

SMOCS relies upon both sensors that can identify and those that cannot. Sensors that can identify drive socially mobile collaboration; those that cannot provide additional contextual data.

3.3.7 Device

A device refers to a smart device controlled by an entity (we assume its owner): it is a device that can run sensor code to gather contextual data about its environment. Dumb devices cannot gather contextual data, so are not instances of our architectural "device" element. Examples of devices include a smartphone or a desktop computer. Some devices are mobile and some are not.

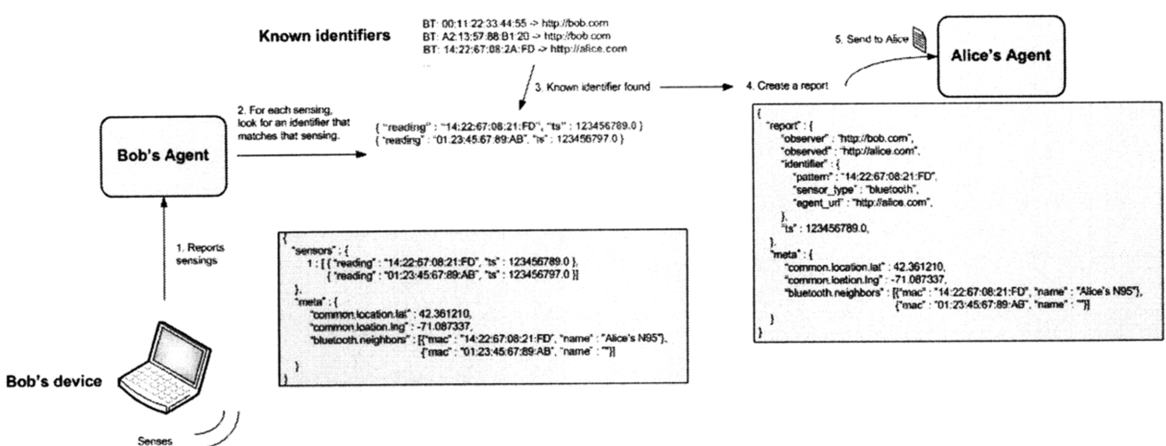

Figure 3-3: Workflow of raw contextual data being transformed into reports and shared with other agents.

Devices can have multiple different sensors. Devices poll their sensors periodically for contextual data and report it to their owning agent over the Internet. If the device is mobile and carried, the sensed contextual data constantly updates the current context of its agent; if not (e.g. a desktop computer) the sensed contextual data is stored but not applied to the user's current context. (After all, the user might not be anywhere near the environment that the desktop is reporting the context of.)

Once sent to the agent, raw contextual data is scoured for identifiers. If another agent has shared their identifier with the agent processing the contextual data, and that agent is identified, a report is created and passed to the identified agent. The report contains information such as the agent that made the identification, which identifier was used and, most importantly, the context of the environment in which the identification was made. This is the key part of a SMOCS system. The workflow is shown in Figure 3-3.

3.4 Architectural schematics

We now present a number of schematics showing how each of the architectural elements interact at different granularities. The structure of agents, relationships and aggregators is shown in Figure 3-4.

Figure 3-5 shows the following visually: an entity is represented by an an agent. An entity has many devices - some may be mobile, others static - and the entity manages these devices from their agent. Each device has many sensors. The sensors sense contextual

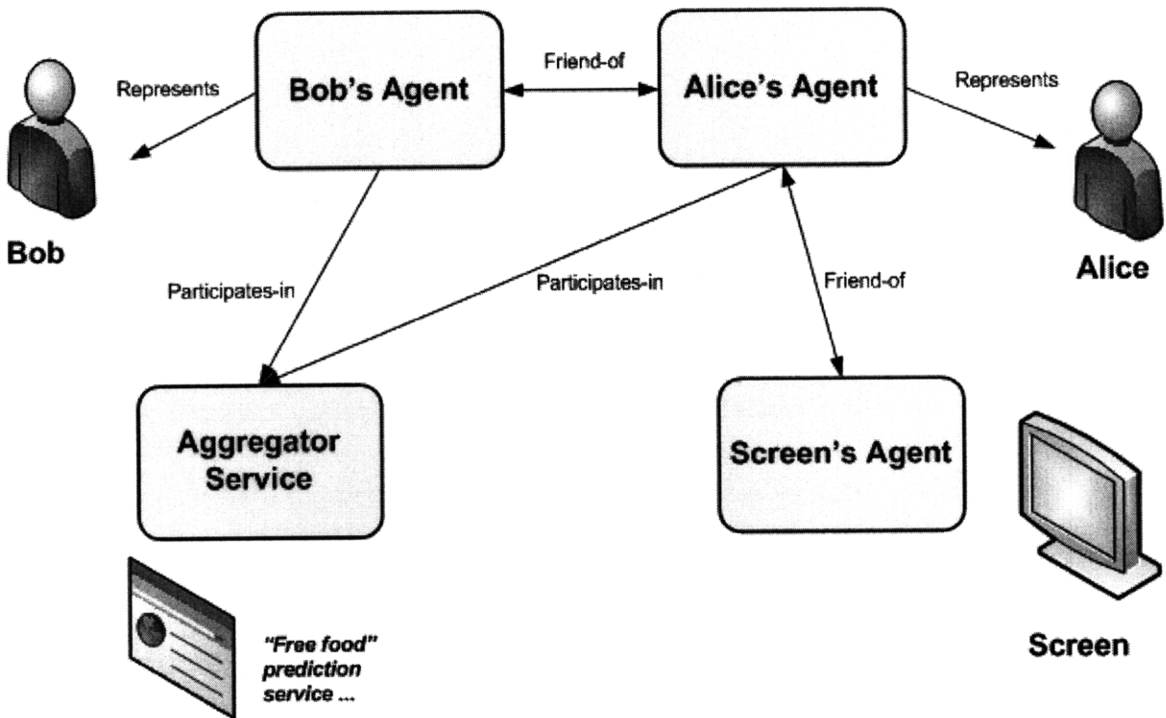

Figure 3-4: Interaction between agents and aggregators through relationships.

data about the environment that they are in. The device periodically polls its sensors, aggregates their readings and reports this data back to its owning agent. Figure 3-3 showed how the agent scours this report for known identifiers. If an identifier is found, the contextual data that was sensed is forwarded on to the identified agent.

3.5 Architectural subsystems

SMOCS makes use of many subsystems in order to fulfill its architectural requirements. These include support for distributed eventing between agents in a publish/subscribe model, interoperability with other social networks, a plugin framework that allows developers the power to extend SMOCS however they desire, and the all-important privacy control subsystem. In this section we briefly overview these subsystems and their architectural properties.

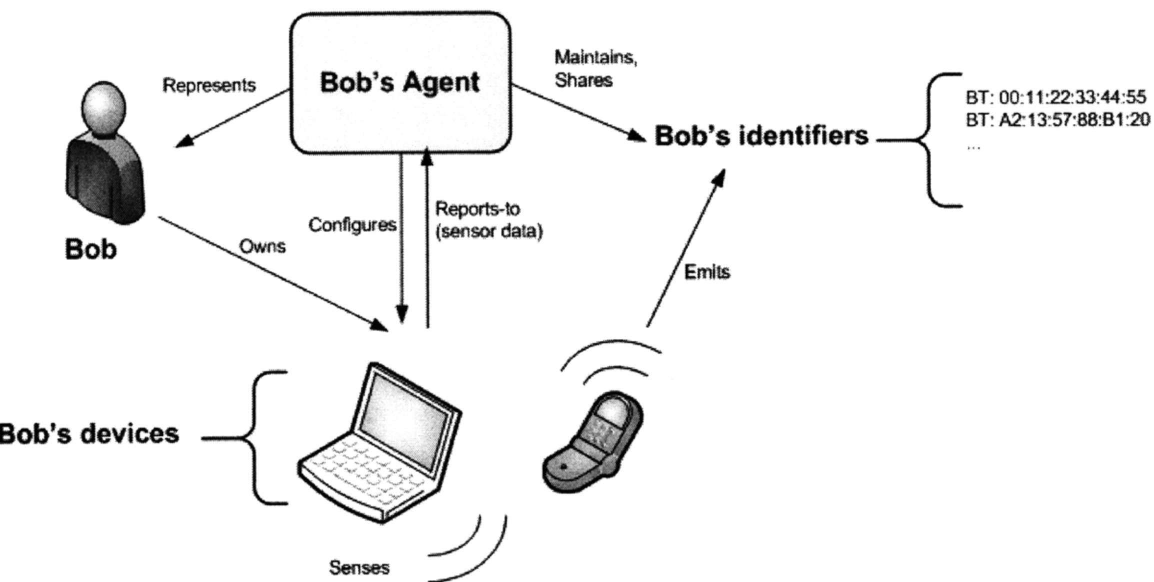

Figure 3-5: Figure detailing the relationship between agents, identifiers, sensors and devices.

3.5.1 Distributed eventing

The architecture of the distributed eventing subsystem follows a simple publish/subscribe model. Users subscribe their agents to particular events on the agents of their friends. When an event is raised on an agent, the agent notifies all subscribers to this event by publishing data to each subscriber's agent. Figure 3-6 gives a conceptual diagram.

Publish/subscribe architectures enable efficient information delivery from creators of data to those interested in the data. SMOCS uses this pattern to enable context-driven applications shared between friends. For instance, Alice's agent could subscribe to Bob's agent to be notified when Bob arrives at work. Alice's agent could react to the publishing of this event by sending Alice an email alert.

The specifications and implementation of the distributed eventing architecture are discussed in detail in appendix A.7.

3.5.2 Network interoperability

To be truly viral, SMOCS is designed to be interoperable with existing social networks. Interoperability allows users to share data with friends using a different social networking platform without requiring those friends to adopt SMOCS. When non-SMOCS users see the value SMOCS provides they may choose to create an agent and join the network. In-

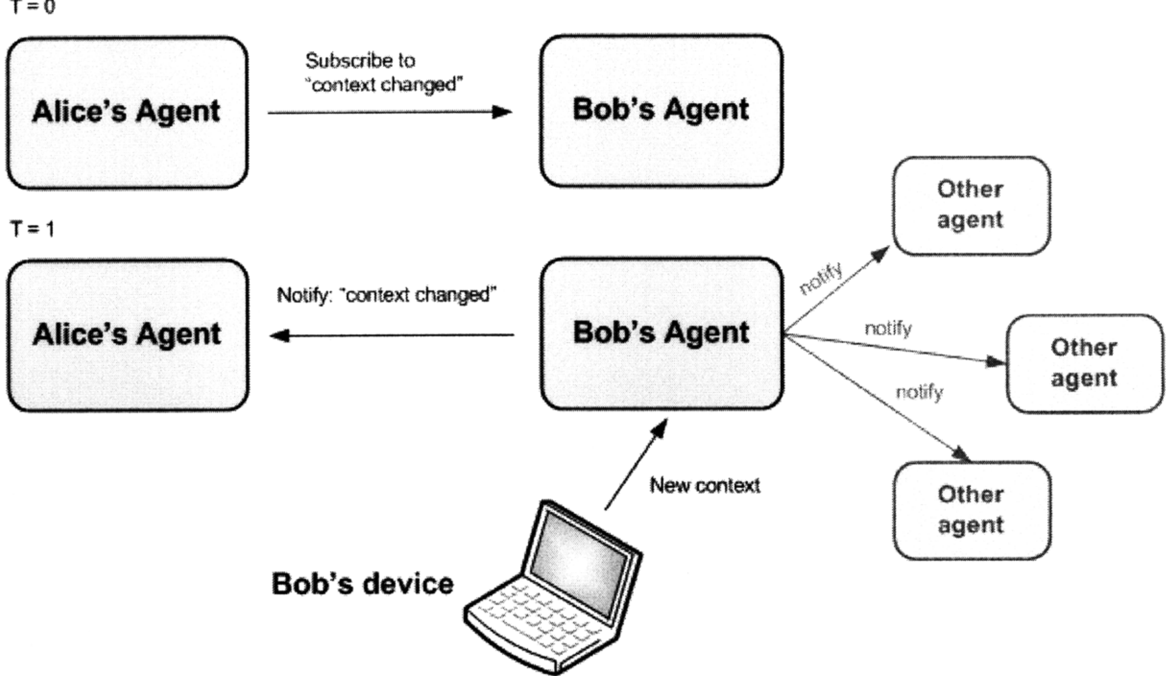

Figure 3-6: Distributed eventing architecture. Alice subscribes to Bob's "context changed" event and is notified when the event is raised.

teroperability thus increases the likelihood of adoption and also increases the immediate value given to users: from the outset, users of the SMOCS architecture can gather their own contextual data and share it with non-SMOCS network users before they are part of a SMOCS network.

Inter-operating with external networks requires well-defined abstractions around the interfaces between SMOCS and the external networks. Each external network has different capabilities and a different implementation is required for each <*network, capability*> pair. We abstract around this implementation variance through the architectural element "network". A network exposes a set of functionalities: each individual network (Facebook, Twitter, etc.) subclasses a common network object and implements each functionality in its own way.

3.5.3 Plugin framework

SMOCS enables contextual data collection and sharing at a large scale. It is impossible for a single implementer to foresee every possible use of this data, thus it is crucial that SMOCS define a clear and flexible plugin framework that can be used by software developers to

extend the functionality of a SMOCS implementation.

A plugin is a small application built on top of SMOCS that adds new functionality for data collection, or that manipulates data stored in an agent in some way. Our architecture calls for sandboxed plugins; that is, plugins should have limited access to data stored on the agent. This is due to privacy concerns: if a plugin can manipulate private data, it is logical to restrict the access of the plugin to a subset of data.

The plugin framework and its API are described in appendix A.8. Example plugins are given to demonstrate architectural extensibility in the next chapter.

3.5.4 Privacy controls

Privacy in SMOCS is defined in terms of data ownership. Ownership comprises physical control over one's data and where it is stored, and the ability to define who has access to it. The distributed agent-based architecture of SMOCS enables control over where data is stored. Access controls on a per-user per-data type basis are provided in order to control access.

Per-user access is relatively straightforward, as the public key infrastructure (PKI) required for authentication and encryption (details in section 3.6.1) means that any agent requesting data from another agent can be authenticated. For instance, if Alice requests data from Bob, Bob is able to authenticate that it really is Alice who is requesting this data. If Bob does not want Alice to have access to his current location, his agent can return an empty response when Alice requests it. This system allows agents to respond differently to requests from different users for the same data.

3.6 Protocols and API

In this section we describe the core protocols and APIs required by the SMOCS architecture.

3.6.1 Protocols

Agents are assumed to be connected via the Internet, and are designed to follow HTTP RESTful design practices. (For a description of REST and our arguments for choosing it as the basis of our protocols, see appendix A.2.) The system provides two sets of protocols:

1. **Entity-Agent protocols** - A RESTful protocol that exposes object-oriented data stored in an agent to client applications (e.g. user interfaces) that the owner of the agent controls. The protocols are the basis for the web-based user interface discussed in appendix A.6.

2. **Inter-Agent protocols** - A non-RESTful protocol for interaction between two agents. This set of Inter-Agent (IA) protocols defines how two agents communicate with each other. It is not purely RESTful due to security issues with standard REST. Details can be found in appendix A.3.

The IA protocols describe the core functionality of the agents, and the EA protocols describe how an owner of an agent can interact with that agent to change settings and manipulate data stored in it.

The decision to make the IA protocols not entirely RESTful was driven by a technical difficulty with security in REST. The IA protocols have two requirements:

1. **Authentication and encryption** - Requests between agents have to be authenticated and encrypted, as different data can be returned depending on the identity of the agent making the request. If a friend of an agent asks for that agent's profile, different data will be returned than that returned to a stranger. This is a relatively simple problem to solve in a centralized website where identity can be verified by the central server, but in a distributed network the problem is more difficult.

2. **Be built on HTTP** - We want any HTTP speaking device to be able to act as an agent so that our architecture survives into the future when devices are smarter, and could even be run on a smart mobile device.

In centralized servers, HTTPS [50] is often used to encrypt client-server communication. HTTPS supports encryption over TLS. Authentication is supported by HTTP/1.0 (basic user name, password authentication) and more complex digest-authentication is supported by HTTP/1.1 [20]. However, both authentication schemes require the user to have an account on the server that they are requesting data from. This is acceptable for our REST APIs, but not acceptable for our IA APIs: for every agent to have an account on their friend's agents would introduce n^2 worth of unnecessary storage overhead in the system.

A public key infrastructure is needed to enable authentication and encryption. In SMOCS, we assume that agents can generate (or provide) their own RSA key pair. Then, requests can be encrypted and authenticated as necessary. For full details of our IA security protocol, see appendix A.3.

3.6.2 API

The IA functions required by agents in the SMOCS architecture are:

- **give** - Gives an identifier from one agent to another. For example, Bob could give the identifier <AA:BB:CC:DD:EE:FF, Bluetooth, Bob> to Alice if Bob wants to be detected by Alice's sensors, and Bob knows that Alice has a Bluetooth sensor.
- **report** - Reports sensed contextual data to the agent. As an example, assume Bob has shared his identifier with Alice as above. When one of Alice's devices sends her agent sensed contextual data and Bob's identifier is found in this data, Alice's agent calls Bob's agent's **report** function, passing along the contextual data her device gathered. This, along with **give**, is the backbone of SMOCS.
- **sensors** - Lists the sensing capabilities of the agent. This function enables friends to decide whom to share their identifiers with; one might choose only close friends with rich sensing capabilities.
- **raw** - Allows friends to query the agent for raw contextual data stored in it.
- **publish** - Allows friends to publish arbitrary events to the agent. Used for distributed eventing.
- **subscribe** - Allows friends to subscribe to events raised on the agent. When an event is raised, the agent will publish to all of its subscribers. For example, Alice could subscribe to Bob's "arrived at work" event. When this event is raised, Alice will be notified; Bob's agent calls **publish** on Alice's agent to notify her.
- **context** - Gets a distilled view of the agent owner's latest context. For instance, this could distill a set of recently reported locations into a single latest location.
- **key** - Gets the public key of the agent. Necessary for the public key infrastructure.

- **profile** - Gets the profile of the entity that owns the agent.
- **friends** - Lists the friends of the agent.

Aggregators require additional functionality:

- **join** - Allows an agent to participate in the aggregator service.
- **leave** - Allows an agent to leave the aggregator.
- **participants** - Lists all the participants in the aggregator.
- **info** - Returns a set of information about the aggregator.

These functions define the core functionality necessary for the SMOCS architecture.

3.7 Object models

In this section we present the objects required by the SMOCS architecture given the architectural elements, properties and functionality being implemented. Some implementation details are given where necessary; generally, details are provided in the appendices. In the following text, object models are named in bold and begin with capitals, e.g. **Agent**. This is to distinguish object models from architectural elements and make clear the relationship between the models.

3.7.1 Agent

Models a known agent. **Agents** are uniquely specified by their URL.

Name	Type	Description
url	string	The URL of the agent.

3.7.2 User

Models the owning entity of an agent. The **User** is the sole user account on the agent and is required by the RESTful APIs for authentication, as well as to support remote login functionality.

Name	Type	Description
first_name	string	The first name of the agent's owner.
last_name	string	The last name of the agent's owner.
username	string	The username of this owner when accessing the agent.
password	string	Encrypted password of the owner.

Profile

Models the profile of an agent. **Profile**s are extensible through **ProfileItem**s.

Name	Type	Description
user	**User**	**User** containing first, last names; user name, etc.
my_agent	**Agent**	The **Agent** that owns this profile.

ProfileItem

A **ProfileItem** enables users to store arbitrary key-value pairs in their **Profile**. This feature is included for extensibility; plugins built on the SMOCS architecture can define their own **ProfileItem**s and perform actions based upon the **ProfileItem**s of friends. **ProfileItem**s can store private or public information.

Name	Type	Description
profile	**Profile**	The **Profile** that this **ProfileItem** belongs to.
key	string	A unique key for this type of **ProfileItem**.
value	string	A value for this type of **ProfileItem**.
private	boolean	Whether this **ProfileItem** should be shared with others.

Relationship

Models a relationship between two agents. **Relationship**s are directional, as described in section 3.3.3.

Name	Type	Description
source	**Agent**	The source of the friendship.
destination	**Agent**	The destination of the friendship.
pub_date	datetime	When the friendship was created.

Key

A **Key** is a 1024-bit RSA key-pair. E and N are the public component; D, P and Q are the private components, as defined in the RSA specification [61]. A **Key** is associated with an **Agent** and is used for authentication and encryption.

Name	**Type**	**Description**
e	long	The public key exponent.
n	long	The multiple of p and q.
d	long	The private key exponent.
p	long	One of the two primes of the private key.
q	long	The second prime of the private key.
owner	**Agent**	The owner of this key.

3.7.3 Subscription

Models the "subscribe" portion of a publish/subscribe relationship.

Name	**Type**	**Description**
event_type	string	The type of the event that is subscribed to.
publisher	**Agent**	The **Agent** who is publishing the event.
subscriber	**Agent**	The **Agent** who is subscribed to the event.

3.7.4 Aggregator

Models an agent's participation in a third-party aggregator service.

Name	**Type**	**Description**
host	**Agent**	The **Agent** that hosts the aggregator.
type	string	The type of aggregator (e.g. "tracker" or "visualizer", etc.)
joined	datetime	The datetime that the aggregator was joined.

3.7.5 Device

Models a smart device controlled by the owner of an agent. The use of private_device_id is explained in appendix A.9.

Name	Type	Description
name	string	A friendly name assigned to the device.
type	string	The type of this device (e.g. "laptop computer")
is_mobile	boolean	Whether the device is mobile or static.
private_device_id	boolean	Secret device id; hash of above fields.

3.7.6 FixedMeta

FixedMeta models fixed contextual data (commonly referred to as "metadata") to include in all contextual sensing reports made by a device. This is used by static devices to report unchanging contextual metadata, for example the device's location. Devices in the MIT Media Lab, for instance, could be associated with **FixedMeta** that adds "Location: MIT Media Lab" to all of their reports.

Name	Type	Description
device	**Device**	The device that this **FixedMeta** is associated with.
key	string	The type of contextual metadata.
value	string	The value of the contextual metadata.

3.7.7 DeviceOption

A device-specific runtime option. For example, the period to wait between scanning for new contextual data. **DeviceOption**s are commonly used by devices with WiFi positioning capabilities to set the Ethernet interface to scan for wireless base station MAC addresses. **DeviceOptions**, like **FixedMeta**s, are simple key-value pairs; this allows maximum extensibility.

Name	Type	Description
device	**Device**	The device that this **DeviceOption** is associated with.
key	string	The key of the device option to set.
value	string	The value of the device option.

3.7.8 Sensor

A sensor on a device.

Name	Type	Description
type	string	The type of this sensor, e.g. "bluetooth" or "location".
identifies	boolean	Whether this sensor can identify agents from an identifier.
gathers_context	boolean	Whether this sensor can gather context.
device	**Device**	The device that this sensor is connected to.

3.7.9 Identifier

An identifier that maps a sensed pattern to a known agent. Recall that identifiers are tuples of the form *<pattern, sensor_type, agent>*; this model encapsulates this structure.

Name	Type	Description
pattern	string	The pattern that identifies this agent (e.g. a BT MAC address.)
sensor_type	string	The type of sensor to use to detect this agent.
agent	**Agent**	The agent that this identifier identifies.

3.7.10 Sensing

A **Sensing** is a raw reading made by a sensor on a device. Sensings are created by **Sensors** that "can_identify".

Name	Type	Description
agent	**Agent**	The agent whose **Sensor** made this **Sensing**.
meta	many **SensingMetadata**	A collection of key-value pairs that describe the context in which this **Sensing** was made.
reading	string	The raw reading that was made, e.g. "AABBC-CDDEEFF" in the case of a Bluetooth MAC.
sensor	**Sensor**	The **Sensor** that made this **Sensing**.
timestamp	datetime	The timestamp that this **Sensing** was made.

3.7.11 SensingMetadata

Contextual metadata associated with a particular sensing. Each **Sensing** has many **SensingMetadata** that describe the context in which the **Sensing** was made.

Name	Type	Description
sensing	**Sensing**	The **Sensing** that this **SensingMetadata** describes.
key	string	A key for this type of contextual metadata.
value	string	The value of the contextual metadata.

3.7.12 Report

A **Report** is generated when a known agent is discovered from a raw **Sensing** by using an **Identifier**. The **Report** models the agent that was observed, the agent that made the observation (the observer), the identifier that was used when making the observation, and the contextual data (metadata) associated with the report. This metadata is a copy of the **SensingMetadata** that was associated with the original **Sensing** that was transformed into a **Report**. Reports are the core of SMOCS. Through reports, agents are alerted that they have been identified by other agents with whom they shared their identifier, and subsequently gather contextual data about the environment in which they were identified.

Alternatively, a **Report** is made when a mobile **Device** that is carried by the user sends the agent contextual data. This contextual data updates the latest context of the agent, and a **Report** is made and stored by the agent.

Name	Type	Description
observer	**Agent**	The **Agent** that made the **Sensing** that was transformed into this **Report**.
observed	**Agent**	The **Agent** whose **Identifier** was observed.
identifier	**Identifier**	The **Identifier** used to transform a **Sensing** into this **Report**.
timestamp	datetime	The timestamp that this **Report** was made.
meta	many **ReportMetadata**	A collection of key-value pairs that describe the context in which this **Report** was made.
sensing	**Sensing**	The **Sensing** that this **Report** was created from. Can be None.

3.7.13 ReportMetadata

Contextual metadata associated with a particular **Report**. Each **Report** has many **ReportMetadata** that describe the context in which the **Report** was made.

Name	Type	Description
report	**Report**	The **Report** that this **ReportMetadata** describes the context of.
key	string	A key for this type of contextual metadata.
value	string	The value of the contextual metadata.

3.8 Implementation

The SMOCS architecture is implemented as two separate components:

- **Ego** - Ego [3] is a distributed social networking platform in which user-owned and controlled agents collaborate to form a communication network. The resulting architecture is graph-based, with nodes representing agents and edges representing the relationships between agents. Data flows across relationships. Ego implements the distributed, agent-based social network required by SMOCS. Ego also provides a plugin framework, allowing development of arbitrary plugins, the distributed eventing subsystem and the framework required for network interoperability. Ego has been developed jointly with Polychronis Ypodimatopoulos.

- **ContInt** - ContInt is composed of two parts:

  1. **Tentacle**. A small application that gathers contextual data from a device's sensors and reports this back to the owning agent.

  2. **ContInt**. A plugin for Ego that implements the mechanisms for receiving contextual data and storing it in the owner's agent. The ContInt plugin handles the sharing of identifiers with friends, the visualization of historical context, and introduces event types that can be used to create other context-driven applications.

Details of these implementations can be found in appendix A.

Chapter 4

Evaluation

In this chapter we evaluate ContInt, the implementation of our SMOCS architecture. We evaluate the agent-based backend implemented by Ego and the ContInt plugin that runs on Ego. The implementation details are described in appendix A. The implementation is evaluated along three dimensions: scalability and performance, architectural extensibility, and privacy.

4.1 Scalability and performance

ContInt is required to be lightweight for it to maintain virality; one of the requirements of the system is the ability for a user to run an agent on their own machine. In this section we evaluate the processor, memory, storage and network usage of an Ego agent running the ContInt plugin under heavy load.

4.1.1 Metrics

Three metrics are involved:

- **Storage** - It is important that agents store as little data as necessary (whilst keeping the data as rich as possible) so that they can be hosted on less-capable machines. The agent of a user running a sensing device receives a lot of data; assuming the device reports its context every 30 seconds, 2880 reports are gathered per day. Achieving $O(n)$ scalability in the number of contextual data reports stored is trivial - one report is stored for each report sent from a device - but minimizing the bytes that each report

requires is more difficult. The SMOCS architecture calls for each **Sensing** object to be associated with a set of **SensingMetadata** objects; see section 3.7.11 for details. Much of this metadata is the same; for example, if a sensing device is stationary, the latitude and longitude will be the same between reports. The ContInt implementation minimizes the number of bytes stored by allowing reuse of existing **SensingMetadata**. This increases processor and disk usage when storing a report, but means that fewer bytes are stored. Our goal is to measure the number of bytes stored on average per day, and extrapolate this figure to estimate a year's worth of data.

- **Network overhead** - The distributed nature of Ego and the manner in which context is gathered, reported and shared means that network overhead should be minimized in order to maximize performance. We do not expect overhead to be massive, but we aim to measure overhead to estimate the order of magnitude of data transfer between devices, agents and the friends of agents in a typical scenario.

- **Processor and memory usage** - To be run on servers of differing capabilities, we have designed our code to have a lightweight processor and memory usage footprint. We measure each of these resources over time.

All tests were conducted on a commodity machine (dual core 2.2 GHz Intel processor, 6 GB RAM, 200 GB available storage) running Ubuntu 9.04 x64. Agents were hosted by apache2 and requests were handled with the mod_wsgi handler, as mentioned in appendix A.1.

4.1.2 Scenarios

We tested two scenarios:

- **Single agent** - In the single agent scenario, a simulated device reports contextual data to an agent every 30 seconds while metrics are collected. This scenario allows us to measure the system properties in the base case of a single user.

- **Multi-agent** - To simulate a server hosting multiple agents and to understand the scalability properties of inter-agent communication better, 50 agents were run on a single machine. Each agent had a single simulated device that reported to it every

30 seconds. Agents participated in bi-directional relationships; these were generated randomly, with each agent having 0-10 relationships.

The simulated devices in both scenarios were the same. They reported location data, Bluetooth neighbors and temperature data. The location data and Bluetooth neighbors were gathered by extending the ONE delay-tolerant network simulator [34]. ONE, designed as a test-bed for delay-tolerant networking algorithms, provides a number of movement models that can be used to simulate human movement. Each node (person) in the simulation was extended to carry a device that reported the person's location in the simulation. In the multi-agent scenario, Bluetooth neighbors were simulated by assigning each node a random identifier at startup and assuming that neighbors within 10 meters of a node were detectable with Bluetooth. In the single agent scenario, Bluetooth neighbors were randomly generated. Temperature was a random number between 0 and 30 (Celsius). Both experiments were run over a one hour period. The user interface of the ONE simulator can be seen in Figure 4-1.

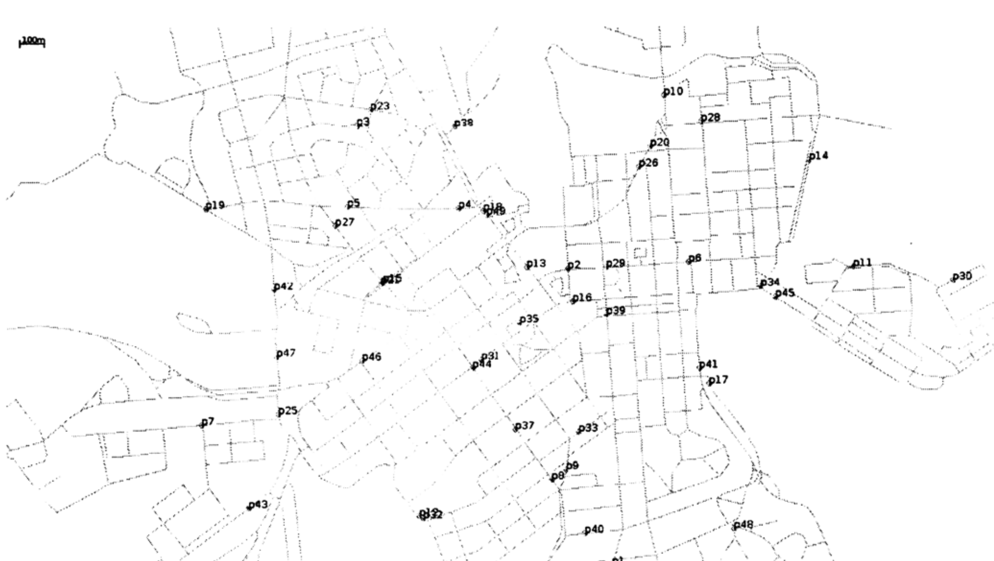

Figure 4-1: The ONE simulator. Simulated people are seen. Each person carries a simulated device that periodically reports location, Bluetooth neighbors and temperature to the ContInt plugin running on an Ego agent.

4.1.3 Results: Single agent

Figures 4-2, 4-3, 4-4, and 4-5 show the disk, network, processor and memory usage observed under the single agent scenario. All readings were gathered by polling the appropriate resource every second over the course of an hour. Further details, where necessary, are given as each result is discussed.

In Figure 4-2 we observe the size of the database of contextual data over time. As we expect, the database grows linearly with the number of contextual data reports made by the agent's device. The final size of the database is 0.5625 MB after one hour; the initial size was 0.1660 MB, so we conclude that 0.3965 MB of contextual data is added per hour. A day worth of data scales this number to approximately 9.516 MB, assuming contextual environment richness stays approximately the same. 10 MB a day translates to 3.65 GB per year per agent - not an unreasonable number considering the capacity and probable future increase of capacity of modern hard disk drives.

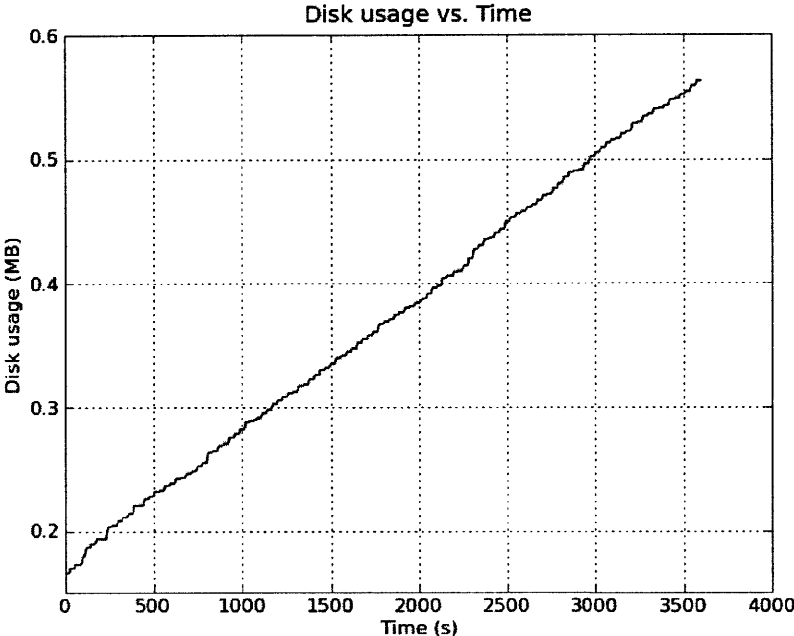

Figure 4-2: Graph showing size of the agent's database over time in the single agent scenario.

Figure 4-3 shows the aggregate network usage of all agents and simulated devices in the system. The y-axis value at any time is the number of bytes transmitted since the previous point. The way that we poll the network means that we observe periods with

no transmission followed by periods with high transmission. Transmissions tend to be in bursts with the most common transmissions being between 1500 and 4000 bytes long. We observe a small overhead; over the course of the hour-long experiment, just 438340 bytes (or 428 KB) are transmitted. This number is not too high for modern smart devices communicating with an agent: GPRS or 3G easily could handle such traffic easily.

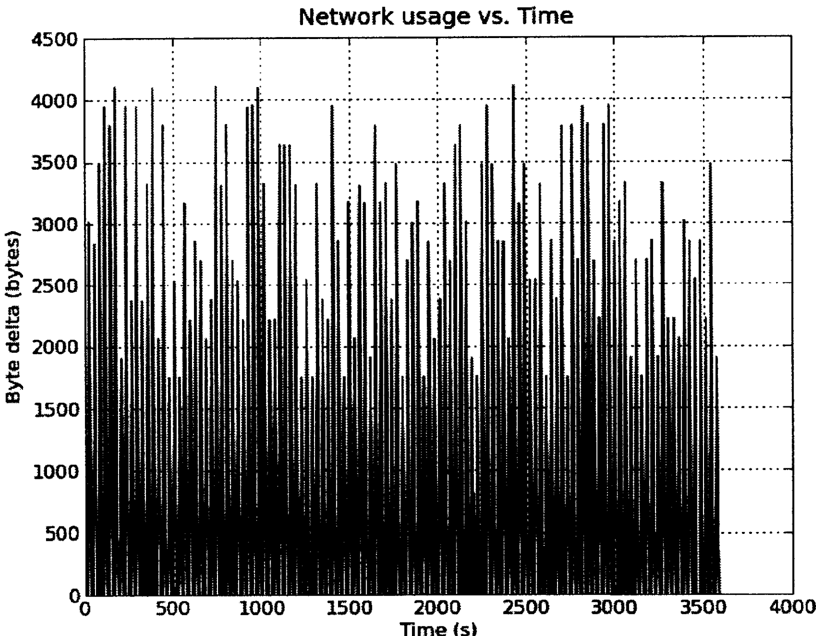

Figure 4-3: Graph showing network usage of all agents and devices in the system in the single agent scenario.

Figures 4-4 and 4-5 show the machine's processor and memory usage over time. Neither of these graphs are particularly telling as the machine used for testing ran a number of background processes and the action of polling and recording the metrics might have incurred additional processor and memory usage.

From the processor usage shown in Figure 4-4 we see a base usage of approximately 8% that can immediately be discounted. The nature of the graph shows moments of peak usage; however, these peaks are infrequent enough and significantly large enough that we attribute them to background work on the machine. In this case, a steady processor usage of between 9 and 35% is observed; or 1% and 27%, discounting the base 8%.

The graph shown in Figure 4-5 shows a constant memory usage of around 1130 MB. The scale on the graph shows that the base 1.129×10^3 is given, and subsequent variation

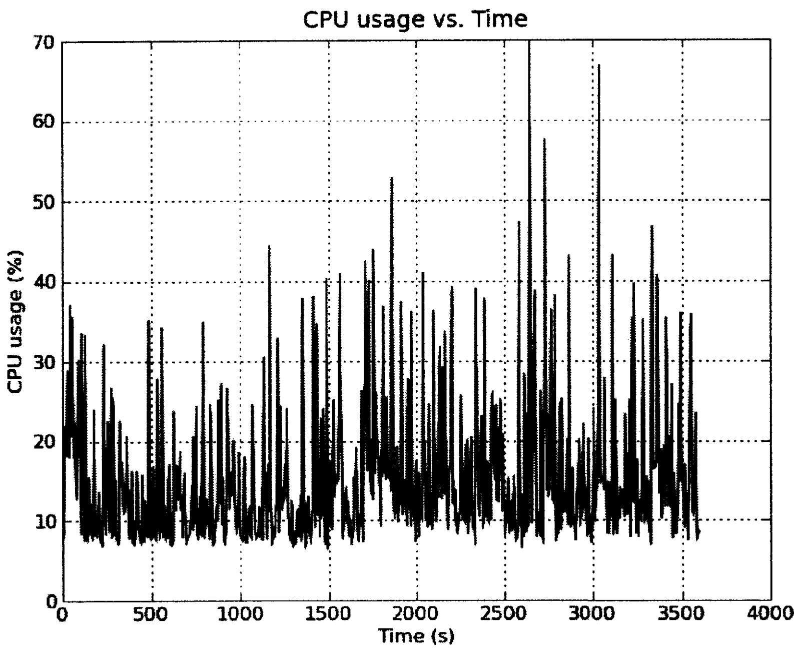

Figure 4-4: Graph showing processor usage of machine hosting single agent scenario.

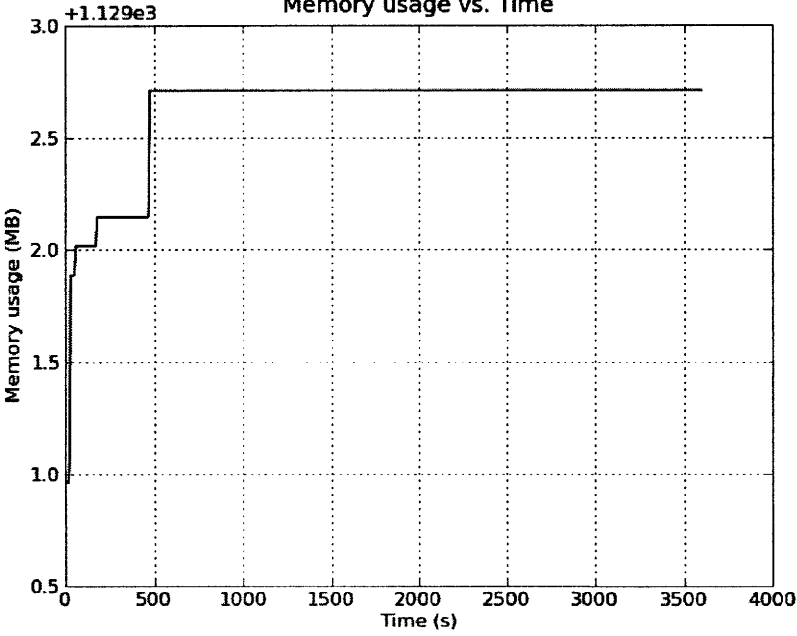

Figure 4-5: Graph showing memory usage of apache2 instances on machine hosting single agent scenario.

is between 1 and 3; essentially, memory usage stays constant. We believe that the memory usage by apache2 is independent of the light load we are putting on it: with 6 GB of RAM available, the server happily takes up 1 GB for its uses. Figure 4-9 is more instructive; it shows the memory usage of apache2 under the multi-agent test.

4.1.4 Results: Multi-agent

Figures 4-6, 4-7, 4-8, and 4-9 show the measured disk, network, processor and memory usage observed under the multi-agent scenario. As in the single agent scenario, all readings were gathered by polling the appropriate resource every second over the course of an hour.

In Figure 4-6 we observe the aggregate size of all agent databases of contextual data over time. Again we observe $O(n)$ growth in the number of reports stored. This is important, because we might expect $O(n^2)$ growth given the sharing of contextual data between users who identify each other and are friends, and in the multi-agent scenario agents have between 0-10 friends. Our explanation is that the $O(n^2)$ factor gets masked by the overwhelming amount of data stored at each individual agent; contextual data is only shared between two friends when their devices meet and this happens relatively rarely. The extra contextual reports being stored by each agent are visible in the final aggregate database size, however: we observe a starting size of 8.3984 MB and a finishing size of 35.3486 MB, for a total of 26.9502 MB of data stored during the test run. 26.9502 is more than fifty times the 0.3965 MB of data collected by in the single agent test; it averages 0.5318 MB per hour. This is what we would expect: agents sharing contextual data gather at least as much contextual data as agents not sharing contextual data. Extrapolating the hourly figure for a day's worth of data we arrive at a figure of 12.76 MB of data per agent per day, or 4.658 GB per year - still a reasonable amount for modern machines.

Figure 4-7 shows the aggregate network usage of all agents and simulated devices in the system. In the single agent scenario we observed 438340 bytes transmitted over the hour; in the multi-agent scenario this number rises dramatically to 369114533 bytes per hour; or approximately 352 MB. The amount of data transferred from each sensing device to its agent is approximately the same as before (31 MB was observed); the rise in network usage is due to the agents sharing contextual data with the agents of other users they identify. 321 MB of network overhead between 50 agents connected by the Internet is not

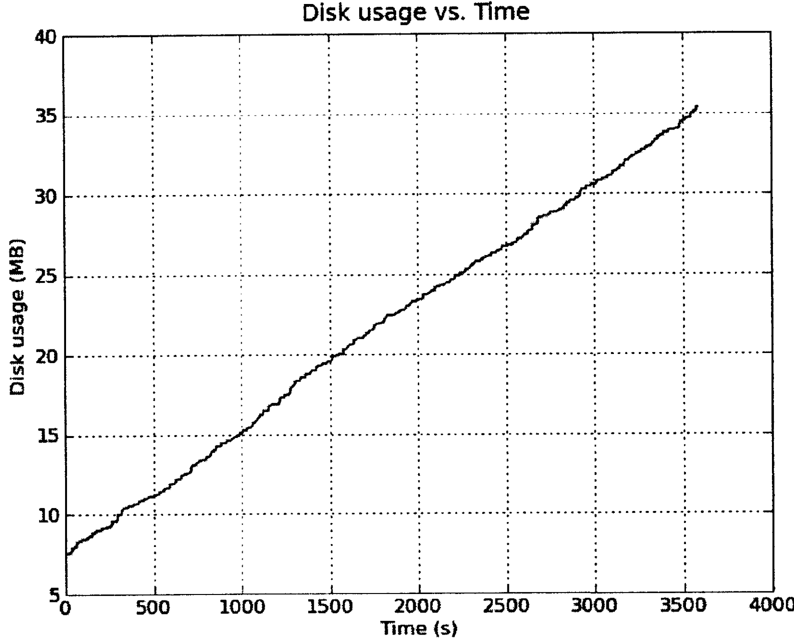

Figure 4-6: Graph showing aggregate size of all agent databases over time in the multi-agent scenario.

a large amount; an average of 7 MB per hour per agent is manageable. Our simulation measured data movement between a clique of friends, so we expect that the 7 MB per hour per agent will not rise significantly in a real-world deployment.

Figures 4-8 and 4-9 show the machine processor and memory usage of the server over time. The processor usage is significantly higher than in the single agent case, with the bulk of the measurements in the region of 30 to 70% and consistently occurring peaks of nearly 100%. The memory usage of the apache2 server rises from a base of 4 GB to 4.4 GB by the end of the test. Again, we attribute the base of 4 GB to apache2's willingness to gather whatever resources it can in order to respond rapidly to requests.

These results are not necessarily indicitative of the actual processor usage or memory footprint of the agent: both metrics are bound to increase as more agents are simulated. Regardless, the results show that a commodity machine can host at least 50 active agents with ease whilst maintaining high availability.

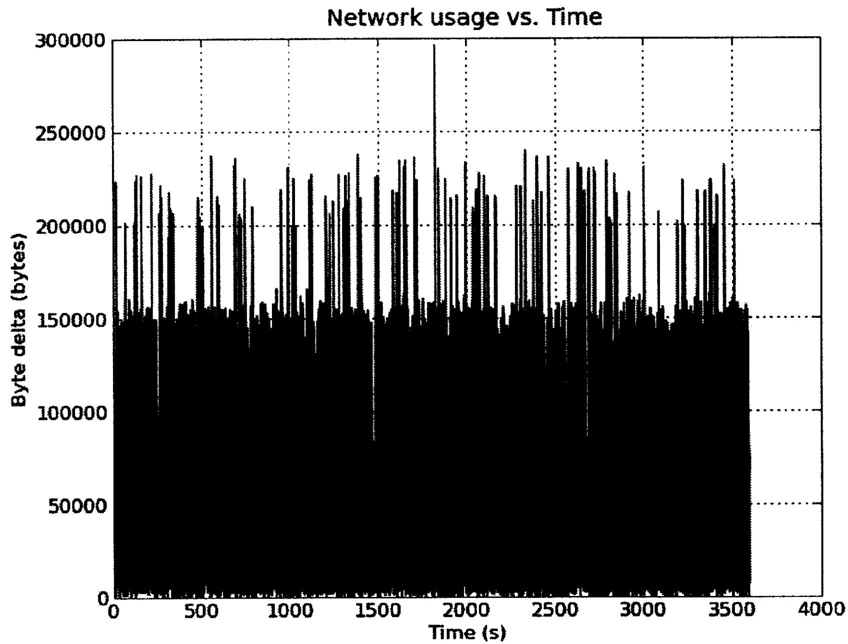

Figure 4-7: Graph showing network usage of all agents and devices in the system in the multi-agent scenario.

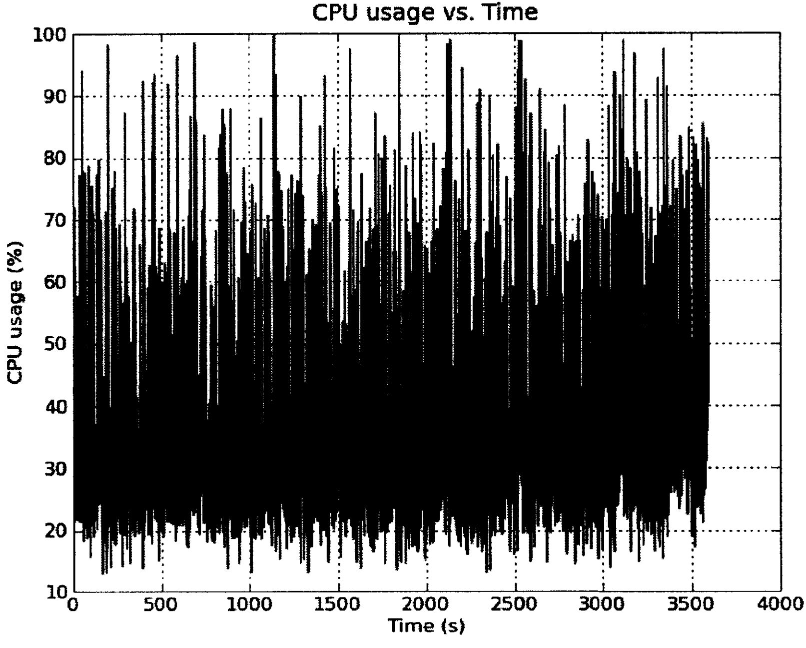

Figure 4-8: Graph showing processor usage of machine hosting multi-agent scenario.

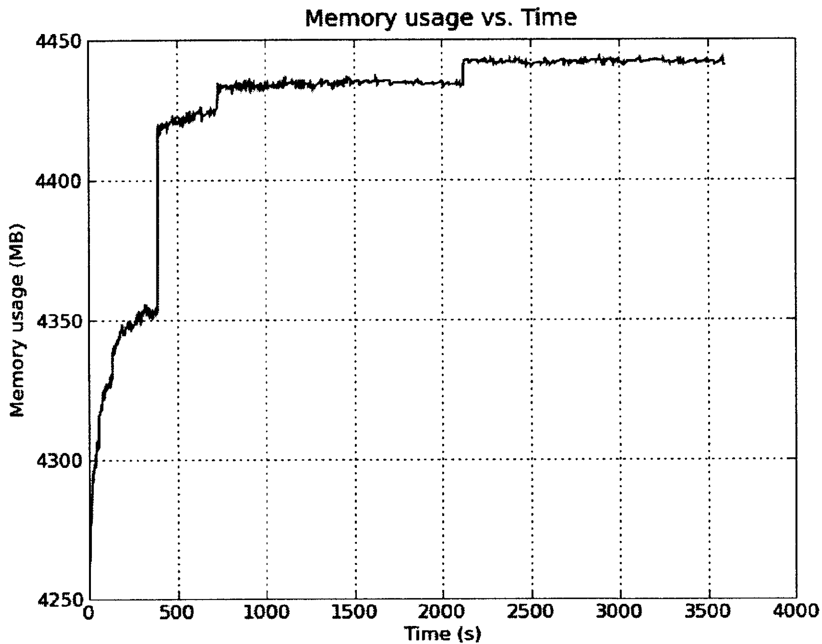

Figure 4-9: Graph showing memory usage of apache2 instances on machine hosting multi-agent scenario.

4.1.5 Discussion

The results presented show the scalability properties of ContInt allow the system to be deployed on commodity hardware with ease. However, various optimizations exist that can be implemented to reduce storage usage and network overhead, and possibly processor and memory usage:

- **Compression** - All contextual data gathered by a smart device is sent to its owning agent uncompressed. The data is then stored on the agent and shared with other agents uncompressed. Compression requires extra processing during compression and extraction (storage and retrieval) but can reduce storage usage. A decision would have to be made as to whether compression is ultimately beneficial, given the increasing size of storage devices compared with the relatively modest increases in processor and memory speeds. Furthermore, uncompressed contextual data is human-readable which makes developing (and debugging) applications that use this data easier. If implemented, this optimization would reduce storage usage and network overhead.

- **Data archiving** - Old contextual data, or data that is of no use, can be archived or deleted. For instance, contextual data reported from a static device owned by Bob that did not describe Bob's context (he was not near the device when the data was gathered) could be deleted. This would reduce storage requirements and, by virtue of removing entries from the database, reduce processor and memory usage during data lookups.

- **Contextual deltas** - In the current implementation, a smart device sends its agent its complete context in every report. If instead the device sent only changes (deltas) that occurred since the last report, network overhead could be significantly reduced. Through use of the system we have observed that contextual data reported by a particular device tends to vary little over time; after all, most devices are carried by people and these people tend to travel from home to work daily. A device carried by a person thus reports the most changes in context when the person moves; the same is true of statically-deployed devices - they observe the most contextual changes as their neighbors change, and this happens when people broadcasting their identifiers move around.

  Reporting only deltas is the optimal way in which to report contextual data if the goal is to reduce network overhead. Consider an 8 hour work day for a user, Bob, carrying a smart device running the Tentacle software. Bob arrives at work, and from then on his device reports that he is at work and the MAC addresses of users broadcasting identifiers around him. In our current implementation, this contextual data is encoded into JavaScript Object Notation (JSON) by the Tentacle software and sent to Bob's agent. A typical example is:


```
{
    "device_id" : 1,
    "sensors" : {
        2 : [ {"reading":"001122334455", "ts":1234567.8},
                                   ...
              {"reading":"AABBCCDDEEFF", "ts":1234568.9} ],
    },
    "meta" : {
        "common.location.lat": "42.361210",
        "common.location.lng": "-71.087337",
    },
}
```

Assuming that Bob's device detects an average of 20 neighboring Bluetooth devices, the JSON string (encoded as ASCII with all whitespace removed) takes up 953 bytes; twenty 42-byte readings (including braces) and 113 bytes for the remainder of the data. If the device reports every 30 seconds, Bob's agent will receive 914,880 bytes of sensor data most of which is redundant. Worse, the contextual data will take up more than 1 MB on Bob's agent; the agent timestamps every report and associates it with the device that made the report.

If instead only deltas were reported by the Tentacle code, the JSON structure would be of the following form:

```
{
    "device_id" : 1,
    "sensors" : {
        2 : [ {"reading":"001122334455", "ts":1234567.8, "type":"start"},
              {"reading":"AABBCCDDEEFF", "ts":1234567.8, "type":"end"}]
    },
    "meta" : {},
}
```

The above example shows how readings could "start" or "end". Any started reading without a corresponding ending is current. Bob's device will make the initial report (now 1253 bytes rather than 953 bytes due to each reading having the added "type" field of size 15 bytes). Assuming that his neighbors do not change until the end of the day, he will send 960 reports (2 reports a minute for 8 hours) with the "sensor" and the "meta" sections reporting no additional data; note the agent requires some sort of periodic heartbeat from the device, else it will assume the device has failed and be wary of using the latest reported context for any context-driven applications (this context could be old). These reports will be 43 bytes; much less than the 953 bytes sent in the current implementation. Assuming that he leaves the office between reports and all readings "end" simultaneously, one final report of 1143 bytes will be sent. The total bytes sent over the course of the work day is then 43,676; over 20 times less data than sent in the current implementation. Some of our assumptions are strong - it is unlikely that there will be no changes in context during Bob's work day - but the figure highlights the order of magnitude difference between the optimality of the current implementation and one that uses deltas.

Such optimizations come at a price, however:

- **Complexity** - Deltas add complexity because the current contextual state of an agent can only be obtained by processing all previous states. Checkpoints can be used to improve the situation: every n reports, an agent could store a checkpoint of its contextual state and subsequently process only reports from this checkpoint onward when determining its current contextual state.

  Another issue is introduced by the need to share reports between agents - that is, the core of SMOCS. When an agent scours reported context for known identifiers and finds one, the agent must share the latest context of the device that made the report with the agent of the identified user. Deltas cannot be shared between agents; the recipient agent does not have the set of prior deltas required to reconstruct the current context. Thus, reports must be constructed with the complete current context of the reporting device. This might be computationally expensive, and means that no network overhead is saved in data transfer between agents - our solution to this problem is thus optimal (aside from possible compression).

- **Processor and memory usage** - Querying an agent for its current state is now more computationally expensive because, as stated in the previous point, agents need to process a set of deltas to construct their current state.

It is not clear that the extra cost of delta processing leads to a net improved performance, though the network overhead between a Tentacle and its owning agent is significantly reduced. This may reduce energy consumption on devices running the software because fewer bytes need to be sent over the network.

The optimizations outlined show how small changes could result in performance improvements. Distributed systems tend to suffer from poor scalability properties due to the overhead required to maintain the network and share data amongst peers. In this section we have shown that ContInt can successfully be deployed on commodity hardware and, whilst certain optimizations could be made to improve its scalability characteristics, the current implementation is efficient along most dimensions. Any optimizations come at a price, and determining whether this price is worth paying is left as future work.

4.2 Architectural extensibility

The extensibility properties of a particular architecture cannot be measured empirically, but they can be asserted through observation of pieces of software built on top of it. In this section we present five diverse applications built on top of the Ego and ContInt implementations and argue that their diverse requirements sufficiently flex the extensibility properties of the architecture. Some of the applications are aggregators built on agents, others are plugins built for agents, and still others are stand-alone applications.

4.2.1 LabTracker

LabTracker is an aggregator that monitors when members of the Viral Communications group at the MIT Media Lab enter and leave the lab. The aggregator allows members of the lab to see when their colleagues were last in the lab or how long they have been there. Historical data is also available; users can see the patterns of their work colleagues. A screenshot of LabTracker can be seen in Figure 4-10.

Viral Communications

who's here?

Charles Amick (in lab since 09:43:05 on Wednesday)

who's not here?

Polychronis Ypodimatopoulos (last in lab 22:40:03 on Tuesday)
Kwan Lee (last in lab ages ago)
Inna Koyrakh (last in lab 18:04:19 on Tuesday)
David Reed (last in lab 16:44:03 on Tuesday)

© 2009 Viral Communications, MIT Media Lab

Figure 4-10: LabTracker user interface, showing when members of the Viral Communications group were last in the lab area.

When a member of Viral Communications makes their agent join LabTracker, they agree to notify the aggregator each time they enter or leave the lab region. This region is specified by the LabTracker; by participating in the aggregator, users agree to install this region on their agent.

LabTracker demonstrates extensibility of:

- **Aggregator requirements** - All aggregators have requirements of some form. Some require certain types of data to be opened up, others require a plugin to be installed

on the agent to expand its functionality. The LabTracker requires that agents create and maintain a geographical region defined by the tracker. Furthermore, the agents are required to report the arrival and departures from this region.

- **Database models** - The aggregator maintains a record of every user's arrival to and departure from a region. This is not bundled functionality, so a new database model must be defined to persist this data. Most aggregators and plugins do this, so we will not highlight it in future examples.

4.2.2 Open Spaces

Open Spaces is a system concept first demonstrated at the MIT Media Lab Sponsor Meeting in April 2009 [25]. The demonstration showed how a user carrying an Amulet [26] - a small device under development by the Viral Communications group that, amongst other things, maintains a user's identity - could interact with third-party "public" computers and use their screens by virtue of being detected by them. The goal was to show how one might take advantage of the processing power in one's surroundings in a more fluid way. Images of Open Spaces in use are shown in Figures 4-11 and 4-12.

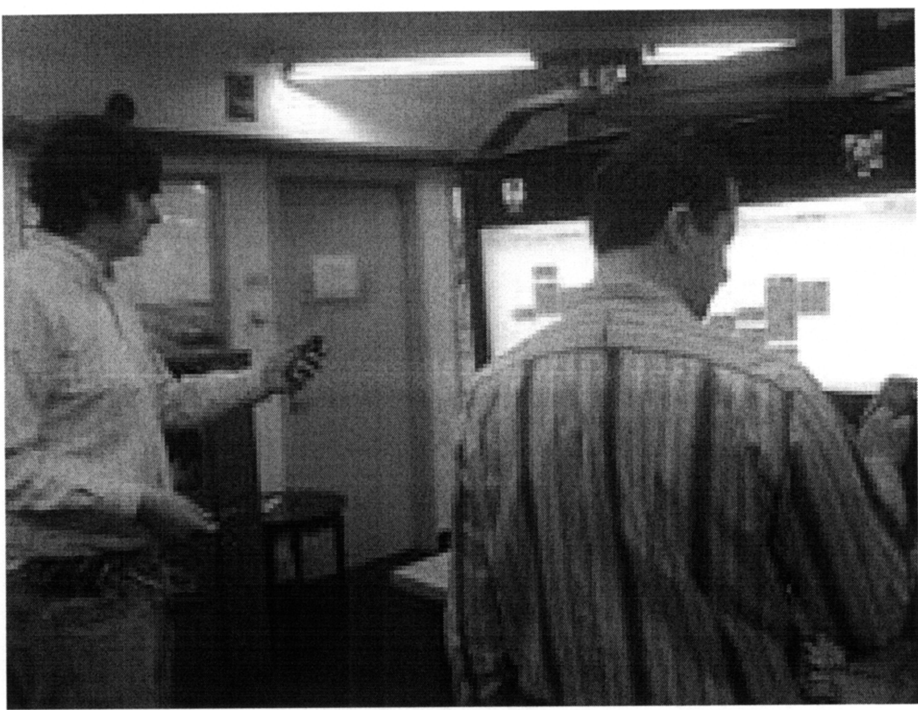

Figure 4-11: Open Spaces being used by two people carrying Amulets.

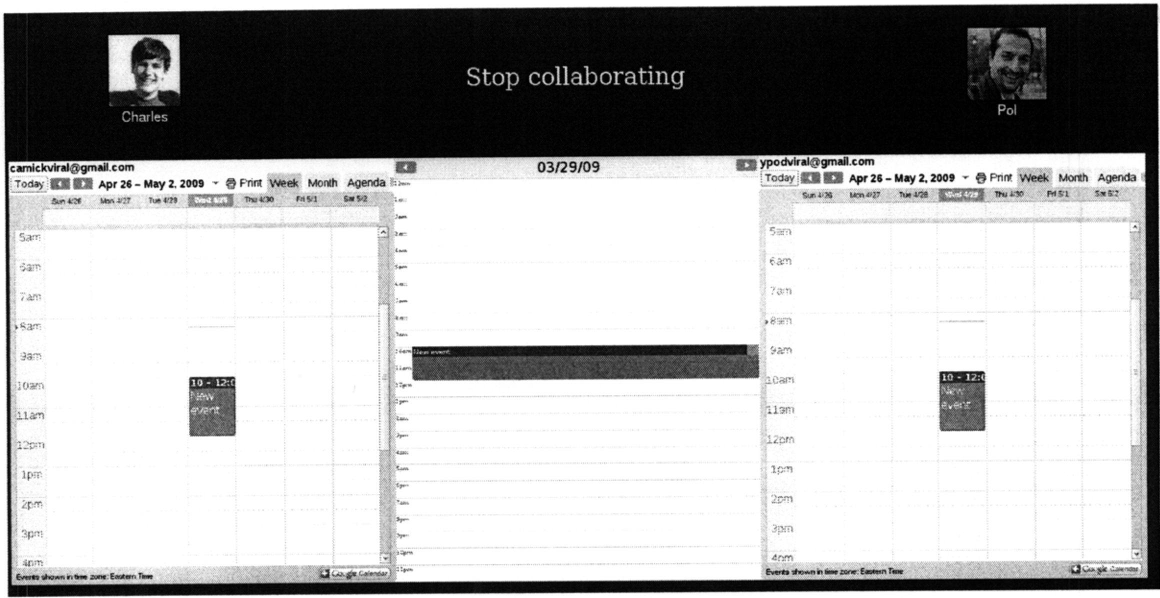

Figure 4-12: Open Spaces demonstration showing the calendars of two users side by side at the screen. The presence of the users was detected with ContInt, and their calendars were retrieved from their Ego agents.

Open Spaces was implemented as a plugin on top of Ego that used ContInt. The main demo screen ran an agent that had one device with two infrared (IRDA) sensors. The sensors were periodically polled for the IRDA ID of the user in front of them. The agent maintained a set of identifiers of the form <(IRDA ID), "IRDA", (User ID)> to map IRDA IDs to user IDs. When a user was detected, a separate process running on the machine powering the screen was notified through the remote eventing mechanism. The mechanism notified the process of the presence of an Amulet and the identity of the owner. That process then retrieved appropriate data from the Amulet and from the agent of the Amulet's owner and rendered it on screen.

Open Spaces demonstrated extensibility of:

- **Sensors** - IRDA was not a sensor type in the ContInt implementation. Open Spaces added this sensor type and the sensor polling implementation necessary for the Tentacle code (the code that runs on smart devices to sense the environment.) The design of identifiers was flexible enough to handle this new sensor type seamlessly.

- **Remote eventing** - The process that handled the retrieval of data from the Amulet and its owner's agent implemented the **publish** method as specified by the SMOCS architecture. The process **subscribe**d to the screen's agent's "Amulet detected" event

and was notified accordingly.

4.2.3 TinyBT

TinyBT (so named for its similarity to TinyURL [46]) is an aggregator that maps Bluetooth addresses to owner agent URLs. Users make their agent join the aggregator if they want to advertise a mapping between their Bluetooth device's MAC address and their agent URL. This can be used by anyone who wants to get the public profile of an unknown Bluetooth MAC address: when a user senses the MAC AA:BB:CC:DD:EE:FF, http://<*TinyBT server*>/AA:BB:CC:DD:EE:FF will map to the URL of the identifier's owning agent, if that agent has joined the service.

TinyBT is also a shortcut for users who want to spread identifiers widely. Rather than sharing their identifiers with specific friends, users can join TinyBT and make their identifier public. Then they can receive rich contextual data from anyone, as anyone can map the identifier back to the owning agent and share detected contextual data.

TinyBT demonstrates how useful aggregators can be and how flexible the APIs that aggregators support are. TinyBT requires users to have the ContInt plugin and, upon joining the aggregator, requires users to select a primary identifier to share. This is another example of aggregator requirement extensibility.

4.2.4 RemoteSense

RemoteSense is a programming framework (and simple console implementation) that allows users to connect to a remote agent and receive contextual data directly from the sensors of the remote agent's devices. Users of RemoteSense must have an agent, and they must provide the framework with their private key and agent URL; the framework can then use these to encrypt and sign communications with other agents. RemoteSense implements the **publish** Inter-Agent endpoint and knows how to **subscribe** to events on other agents. The remote event-driven process described in Open Spaces (section 4.2.2) was implemented with RemoteSense. Figure 4-13 shows the console-based user interface of the RemoteSense application.

RemoteSense demonstrates the extensibility of ContInt's APIs. The choice to use HTTP-based protocols meant that any system that can communicate over HTTP can communicate

```
camick @ newguy: ~/Desktop python remotesense.py http://ego.media.mit.edu/ypod/ http://ego.media.mit.edu/camick/ ~/priv.key
##############################
### 1240929798.55 Connecting...
### 1240929798.59 Connected to: http://ego.media.mit.edu/ypod/ as http://ego.media.mit.edu/camick/
### 1240929798.68 Opened data stream. Active devices:
        n95 (Cell phone, carried)
        ubuntu (Desktop computer)
##############################
### 1240929812.51 Sensing:
        Device: n95
        Context:
            common.location.lat: 42.00856455
            common.location.lng: -71.09816453
            bluetooth.neighbors: ['7F6FF5B148']
            irda.neighbors: []
### 1240929813.51 Sensing:
        Device: n95
        Context:
            common.location.lat: 42.00856189
            common.location.lng: -71.09662053
            bluetooth.neighbors: []
            irda.neighbors: []
### 1240929839.41 Sensing:
        Device: n95
        Context:
            common.location.lat: 42.00063999
            common.location.lng: -71.09934512
            bluetooth.neighbors: []
            irda.neighbors: []
### 1240929862.82 Sensing:
```

Figure 4-13: RemoteSense application showing a stream of contextual data from sensors run on devices of a remote agent.

with an agent. RemoteSense was not envisioned from the outset, but the SMOCS architecture proved able to accommodate it.

4.2.5 Advanced privacy controls

Ego (with ContInt) has been used as a development platform by a number of people since its inception. One researcher developed privacy controls more advanced than the simple ones provided in Ego and (at the time of writing) is in the process of integrating them into the Ego core. ContInt can take advantage of these new privacy controls, as it is built on Ego.

The new privacy controls allow users to:

- **Perturb data** - Users can share perturbed data with peers. For instance, if Bob does not want to tell his friends exactly where he is but would like to give them an approximate region, the privacy controls can be set to perturb his location.
- **Compute functions over a set of encrypted data** - Fully homomorphic encryption [22] is used to compute simple algebraic functions over encrypted data. The scheme allows arbitrary addition and multiplication functions to be performed on encrypted data by a single aggregator that returns a result only decodable by the agents who

provided the data. This can be used by aggregators to collect community-wide intelligence from a group of participants but not actually allow the aggregator access to the raw data or the results.

These controls demonstrate the extensibility of the Ego core. Ego implemented the core of SMOCS, and these new privacy controls were added to the core of Ego with minor implementation changes required. The SMOCS privacy specification of selecting the appropriate data to respond to a given requester was extended with new algorithms.

The above examples are five instances of applications supported by the SMOCS architecture and its implementation. Each application has unique requirements that test the extensibility properties of the system. From these successful implementations and the observation that few changes needed to be made to existing Ego/ContInt code, we conclude that the system has excellent extensibility properties. Future applications will continue to test the SMOCS architecture but, given past and present experience implementing applications we believe that the system will be able to support many new applications in years to come.

4.3 Privacy

We argue that the privacy mechanisms provided by ContInt are sufficient: users control their data. To demonstrate this, consider two opposite cases and in an intermediate case of user privacy requirement.

Alice runs a SMOCS agent on a public hosting service. She shares identifiers with TinyBT in order to spread her identifiers widely so she can gather rich contextual data. She opens up her current context to the public. She shares all her data and is not worried about the privacy implications.

Bob is concerned about his privacy and shares his data with no one. He runs a SMOCS agent on his own machine to which only he has physical access. He shares no identifiers, and instead runs a Tentacle on his smart device. (If he has no smart device, he cannot use the system to gather contextual data.) The Tentacle gathers contextual data and sends it (encrypted) directly to his agent. Bob's public profile shares no information. He is perfectly happy with his privacy, and is in control of his data.

The more interesting case occurs in the middle ground. Assume that Charlie wants to share his context with Alice. Charlie's agent allows Alice's agent access to this data, and Charlie must trust that Alice won't release it to others. This is where notions of information accountability come into play: Alice must accept that she is accountable to Charlie for her use of Charlie's data. Information accountability schemes are difficult; any human-readable data can be copied, and so even software-level controls around data sharing can be compromised. In the future such schemes may be possible; for now, we assume that Charlie must place his trust in Alice.

Our system achieves sufficient privacy because it empowers users to control their data. Everything is opt-in: if users want to gather rich contextual data by virtue of their identifiers being sensed, they must share these identifiers with trusted peers. If users want to get group-wide intelligence, they must share some personal data. If users want to be discoverable in a directory, they must share their name and agent URL. But it is entirely possible to be like Bob and not share anything. Thus, data control is with the users and our privacy requirements are satisfied.

Chapter 5

Conclusion

In this chapter we give an overview of the work presented in this thesis and suggestions for how this work can be taken forward.

5.1 Overview

In this thesis we define the concept of "socially mobile collaborative sensing" (SMOCS) and demonstrate its usefulness as a truly viral contextual data gathering system. We show how users with smart devices and dumb devices can collaborate to gather rich contextual data about their environment in real time. The need for contextual data gathering is motivated by the observation that more and more users are gathering contextual data with static and mobile devices in order to use context-driven applications.

SMOCS, and the practical need for flexible privacy controls, led us to propose a distributed network composed of user-controlled "agents" connected by the Internet. Agents form relationships in order to share data, including the identifiers required by SMOCS. Agents act as personal data repositories and are designed to be as flexible as possible in the data types they store in order to guarantee architectural extensibility. Agents are assumed to be hosted on the Internet and identified uniquely by URLs.

Our SMOCS implementation called for HTTP-compliant RESTful APIs where possible. Entity-Agent and Agent-Agent (or Inter-Agent) protocols are specified; EA protocols are fully RESTful and IA protocols are as RESTful as possible - security requirements forced us to design a non-standard RESTful protocol.

The SMOCS architecture is implemented with two components: Ego, a distributed so-

cial network that handles the core agent-based communications; and ContInt, the contextual data gathering plugin for Ego. We refer to this entire implementation as ContInt, for simplicity. We also present ContInt's device code that performs the actual sensing of environment, the Tentacle. The agent is implemented as a set of web services, with a standard HTML user interface for user manipulation and exposed REST APIs providing the possibility for future, non-browser based interface implementations.

We evaluate ContInt by considering scalability and performance metrics, its extensibility properties and we consider that its privacy controls are sufficient to give users confidence.

We observe the expected $O(n)$ storage use in the number of contextual data sensings stored, and show empirically that the constant (number of bytes per sensing) is low, which is important for scalability. We observe that an agent running a sensor in a contextually-rich area creates on the order of 12 MB of data per day. We assert that uninteresting contextual data (e.g. data gathered by a static sensor that does not apply to the owner of the agent) can periodically be archived to reduce database size, and that 3-4 GB of data per year is not overwhelming for modern systems.

We monitor processor and memory usage, and demonstrate that up to 50 agents can run concurrently on a single, commodity machine (2.2 GHz dual core, 6 GB RAM, Ubuntu 9.04 x64) even at peak usage. Peak usage was simulated by having all 50 agents share data with their friends (a randomly selected subset of the other agents) and receive contextual data inputs from simulated devices periodically. Our observations lead us to conclude that agents are lightweight enough to be deployed on commodity hardware.

Five examples of applications built on top of (or in to) ContInt are assessed to demonstrate the architectural extensibility of the system. We see that the decision to rely on HTTP as a transport protocol and RESTful design principles enables rapid development of diverse applications.

Finally, we revisit privacy with a practical observation of two extreme cases and an intermediate case: a user who demands complete privacy and data control, and a user who demands easy contextual sharing, rich contextual data gathering and less privacy, and a user who is only prepared to share data with a trusted friend. All three cases are satisfied by the system, primarily due to its opt-in structure: no data is exposed without the permission of the user.

5.2 Future work

We expect that future work will focus primarily on reducing the need to share identifiers; in essence, killing the social nature of collaborative sensing. In future years, mobile devices may become more open to developers, paving the way for fewer "dumb" devices and allowing all users to run Tentacle code on their personal device.

Such developments might seem to negate the usefulness of SMOCS. While it is true that the need for collaboration and, in particular, sharing personal identifiers with peers may be reduced, we believe that the agent-based architecture used by SMOCS can be used by future smart devices.

Immediate problems to be solved in order to secure SMOCS adoption include agent migration between URLs and the inclusion of information accountability mechanisms:

- **URL migration** - We have relied upon URLs in our implementation as unique identifiers. When an agent wants to change its URL the process is difficult as the URL must be updated in a multitude of places.
- **Information accountability** - We are interested in tracking information flow around the network, and implementing preliminary accountability methods so that agents can report their use of another agent's data back to that agent. This will allow users to be even more in control of their own data.

Note that these problems are do not prohibit deployment. As of writing, more than 30 people are actively using Ego/ContInt in the MIT Media Lab. We expect to see this number rise as functionality is added.

Our goal with SMOCS is to demonstrate one possible way in which people could use the computing resources in the world around them for their benefit. This as an instance of the Third Cloud principle: the first cloud (the Internet) and the second cloud ("Web 2.0") will be displaced by a third cloud of hyper-local resources collaborating to achieve a task [49]. We expect that most future work will focus on other applications of this principle. Such work may not build directly upon SMOCS, but we expect that there will be much to learn and develop from our architecture and its evaluation. The offloading of data collection, processing, storage and sharing due to our agent-based approach proves to be flexible and powerful.

We hope to see future work that makes better use of the idling computational resources in the world around us, especially in the hyper-local world as portable devices increase in capability. As Henry Ford said: "Coming together is a beginning. Keeping together is progress. Working together is success." In this thesis we have shown how devices can come together and work together to achieve success. We expect progress.

Appendix A

Implementation

In this appendix we detail ContInt, our implementation of the SMOCS architecture. ContInt is built as a plugin for Ego, so many details concern work done on the Ego project. We present details of Ego, the ContInt plugin, and the Tentacle code that gathers contextual data from a device owned by an agent. These three components were first presented in section 3.8.

We begin by presenting the decisions made that determined our implementation tools. We then detail various aspects of the implementation that may be of interest to the reader, including the distributed eventing subsystem, an overview of RESTful design patterns, security frameworks and our plugin framework.

Developer-level technical details are available from the Ego documentation online at http://enchalla.media.mit.edu/ego/.

A.1 Implementation decisions

The requirement of the Ego agent to be personally maintainable, reachable across a network and to reuse standard protocols led us to implement the agent as an intelligent web service. A web service architecture was chosen because:

- **Internet** - The Internet was the network to use for data sharing.
- **Abstraction** - A web service-based implementation forces abstraction between implementation and API usage; data passed across HTTP must be serialized in a particular way, breaking any language-specific implementation details. This separation

is crucial in software design.

- **Protocol reuse** - Web services allowed us to build upon the existing HTTP standard and RESTful design patterns (see appendix A.2 for details.) Using established standards enables developers to build upon the system more easily.

- **Caching** - Existing network infrastructure is highly optimized for delivering data across HTTP; memcached [40] and other caching software could be used by the agent without it having to implement its own, domain-specific cache.

- **Outsourcing hosting** - A requirement of the SMOCS architecture was that users should be able to run their own agent or have one hosted for them. Implementing the agent as a web service enables users to run their own easily or to have it outsourced to a web hosting company. Our implementation is compatible with the Apache web server [17], meaning that it can be hosted by any machine running an Apache instance. It is also compatible with the Web Server Gateway Interface (WSGI specification [13] so can be run on any server with WSGI bindings. As of writing, WSGI has wide support: Apache and Microsoft's Internet Information Services (IIS) [11] server support it [62].

Having chosen to implement the agent as a web service, a web application framework and set of languages for implementation had to be chosen. We chose to implement agents in Python [19] on top of the Django web framework [18]. Python was our language of choice because:

- **Portability** - Python can run on every major operating system and a variety of devices, from high-end web servers to cellular telephones. Our implementation of the agent can run on a portable Nokia N810 Internet tablet.

- **Rapid development** - Python allows developers to express themselves concisely. Many common design patterns are abstracted, so more can be done in fewer lines of code.

- **Active community** - The Python community is large and active. Many common problems have been solved elegantly by others.

- **Libraries** - Python has thousands of extension libraries and frameworks that can be used. We used Django, geopy, httplib2, urllib2, pyyaml and simplejson.

Django is a web application framework built in Python that enables rapid development of database-backed websites. Django, Pylons [47] and web.py [56] were compared and Django was chosen, primarily due to past familiarity with the framework, but also because:

- **Community** - Django has the most active community of any Python web framework.
- **Reliability** - Django is a proven framework used by a number of large websites, including the Lawrence Journal-World newspaper and its affiliates.
- **REST support** - Django has a number of third party plugins for building both RESTful and RPC-based applications. It is also bundled with essential tools for HTTP security, and prevention of Cross-Site Request Forgery (CSRF) attacks.
- **Abstraction** - Django implements a Model-View-Controller paradigm (Model-Template-View in Django-lingo, but the same as MVC) that allows for easy separation between data models, manipulation of models and views of the models.
- **Object expressiveness** - Django has a mature object-relational mapper (ORM) for persisting Python objects to database tables. The database drives the agent, so a mature ORM is essential in order to express complex object relationships. Django's ORM supports one-to-many and many-to-many relationships as well as object inheritance.

Agents acquire large quantities of data over their lifetime and this data must be persisted in a database. As noted, Django has a mature database wrapper. The wrapper supports multiple database servers, including MySQL [2], PostgreSQL [24], and SQLite [9]. SQLite was chosen because it is a self-contained file-based database. MySQL and PostgreSQL would both require separate database services to be run. If this was required of users the system's virality would decrease as the barrier to entry increases. SQLite is an excellent choice for a personal, lightweight web service like the agent for a number of reasons:

- **Single file** - The database is stored entirely in a single file on the file system. It can be backed up simply by copying it elsewhere. Backups are critical to a system that

gathers sensitive data; we do not want users to lose all their personal data to a server crash. Thus it is important to make the backup process easy. SQLite makes full database backups trivial.

- **Permissioning** - If the database was run on a centralized database server, permissioning would have to be needlessly replicated from the file system. Any time permissions are replicated, the possibility of error is introduced.

All web-service components of Ego and ContInt follow the same architecture. That is, both the aggregator and the agent are implemented in this manner. Aggregators are implemented as agents with added functionality; see section 3.6.2 for details of the added functionality.

A.2 RESTful design patterns

Where possible, we implemented our communication protocols using the REST architectural style. We evaluated XML-RPC (eXtensible Markup Language Remote Procedure Call) [30], SOAP (Simple Object Access Protocol) [27] and REST (Representational State Transfer) [15] as alternatives for web service protocol implementation. We chose REST over XML-RPC and SOAP for the following reasons:

- **Object-orientation** - REST takes an object-oriented approach to web service design. "Resources" are addressed with URLs, and manipulated with standard HTTP verbs. XML-RPC and SOAP are function-oriented; rather than asking an object to update itself, the developer must write a function that takes an object and a set of updates and applies those updates to the object in a functional manner.

- **HTTP verb reuse** - REST reuses standard HTTP verbs (POST, GET, PUT, DELETE) for the standard CRUD (CREATE, READ, UPDATE, DELETE) methods needed to manipulate database-backed objects. In XML-RPC and SOAP, one has to write a createObject, getObject, updateObject and deleteObject method for each type of object being manipulated; the power of HTTP's built-in verbs is lost.

- **Broad support** - REST tends to be the web service implementation of choice for most modern "Web 2.0" websites. Flickr, Twitter, Facebook and Google provide RESTful

APIs. SOAP tends to be used in corporate environments because of its support from Microsoft services and its static typing capabilities.

- **Language agnostic** - REST can be used by any language that provides HTTP bindings. XML-RPC and SOAP need XML parsers at the very least; utility libraries would be extremely useful due to XML-RPC and SOAP's verbosity.

REST has some disadvantages:

- **Loose protocol** - Beyond the architectural and naming conventions, REST offers little in terms of protocol. Message bodies (data sent in requests) can contain arbitrary data; likewise, any data type can be returned. Some web services serialize data as XML, others as JSON and still others as YAML. To be as flexible as possible, many services support content negotiation [16]; requesters provide an Accept: (value) key-value pair stating the serialization method to use in the server's response.

- **Error reporting** - REST indicates errors through HTTP status codes. The status codes for errors include 400 (bad request), 401 (unauthorized), 403 (forbidden), 404 (not found) and 405 (method not allowed). These codes alone do not provide enough information to debug a complex problem. XML-RPC and SOAP have a well-defined error structure in which error codes and messages are formatted in a particular way. RESTful services tend to define their own error reporting format and enforce it by convention. This can be irritating, as each service has a different error reporting implementation. For example, Facebook and Yahoo!'s REST protocols have different error reporting standards [14, 33].

We believe the advantages of REST far outweigh its disadvantages. Ego uses REST extensively to export objects modelled in an agent's database for manipulation over HTTP. This enables data access and manipulation from anywhere: agent user interfaces could be built for any device from an iPhone to a Desktop computer. Our web-based user interface, detailed in appendix A.6 is built on the exposed REST interfaces.

A.3 Inter-Agent security protocols

Because Ego is a distributed network formed by a society of agents, it is necessary to define clear inter-agent (IA) communication protocols. The protocols allow agents to maintain the

structure of the network and share data with each other. Examples of inter-agent communication include agents requesting each other's profile information, forming relationships and alerting each other of events.

The design of the IA communication protocols was driven by two key motivations. The protocols had to:

1. **Support authentication and encryption** - Different data could be returned depending on the identity of the requester. If a friend of an agent requests the agent's profile different data will be returned than that returned to a stranger asking the same question. This is a relatively simple problem to solve in a centralized social website where identity can be verified by the central server, but in a distributed network the problem is more difficult.

2. **Be built on HTTP** - We wanted any HTTP-speaking device to be able to act as an agent. Furthermore, we wanted to be able to communicate with an agent from any HTTP-speaking device, even if that device could not run an entire agent.

In standard centralized servers, HTTPS [50] is often used to encrypt client-server communication. HTTPS supports encryption over TLS. Authentication is supported by HTTP/1.0 (basic user name, password authentication) and more complex digest authentication is supported by HTTP/1.1 [20]. However, both authentication schemes require the user to have an account on the server that they are requesting data from. This is acceptable for our REST APIs which are designed only to be accessible by the owner of the agent, but not acceptable for our IA APIs: for every agent to have an account on their friend's agents would introduce n^2 worth of unnecessary storage overhead in the system, and be very difficult to maintain.

A public key infrastructure (PKI) was needed to enable authentication and encryption. PGP [6] is a distributed PKI system that allows users to maintain, share and endorse each other's public keys. PGP could be used by the agents, but introducing another third-party component to the architecture reduces its virality by increasing the conceptual overhead requirement for new users.

In our solution, agents generate their own RSA 1024-bit key pair the first time they are run. The process is transparent to the user. Agents advertise their public key at a well-known address so that others can access it easily. We then use the public key for two

purposes:

1. **Encryption** - When making a request of a remote agent, the remote agent's public key is retrieved and the request is encrypted with this public key. Only the remote agent can decrypt the request by using its private key.

2. **Authentication** - It is necessary for the requesting user to be authenticated in order to access the information they want; agents can respond to the same query with different results depending on the identity of the asking user. In the header of a request, the following three fields are passed:

   - **ts** - The UTC timestamp of the request, in epoch-seconds.
   - **req_agent** - The URL of the requesting agent.
   - **sig** - The cryptographic signature over the entire request made using the requesting agent's private key.

In addition to the core three fields, the header also includes whatever data is needed by the requested method. For instance, if Alice was requesting Bob's latest context, her request might contain a set of **filters** declaring which types of context to return.

The entire request is encrypted before being sent to the target agent. Because HTTPS cannot be used, one cannot use a standard HTTP POST with individually encrypted fields. Instead, the entire request is bundled into a JSON (JavaScript Object Notation) object and encrypted. This encrypted JSON object is then passed in the HTTP POST field "JSON". When the remote agent receives the request, it decrypts the posted JSON object and attempts to authenticate the identity of the requesting user.

The **ts, req_agent** and **sig** are deserialized from the decrypted JSON object. The public key of the **req_agent** is looked up in a cache; if it does not exist it is retrieved from the URL specified by **req_agent**. Once obtained, the key is used to verify the integrity of the data by authenticating the signature against the data. Finally, the timestamp **ts** is compared to current UTC time. If the time is within a small delta then the authenticity is verified and an appropriate result is generated and returned.

Because responses are given in the form of HTTP responses, the request is vulnerable to replay attacks. Snooping attackers can observe an encrypted request passing from Alice to Bob. The attacker could capture the request and replay it at a later date; the request, after

all, is a valid one - it has the required signature of Alice and has not been tampered with. Replay attacks are thwarted by the inclusion of the timestamp of the request in the field **ts**. If a request is received and the delta between the request's timestamp and the current time is over some limit, the request will not be processed and an error returned. This requires agents to have synchronized clocks. We implemented clock synchronization using the Network Time Protocol (NTP) [44]. Agents maintain a local clock offset from UTC, and their local time must only be updated every month or so; qualitative observations have shown minimal drift thus far.

The IA security protocol workflow is shown in Figure A-1.

A.4 Naming conventions

Naming conventions are given to improve modularity and understanding. The following rules are adhered to:

Relative URL	**Description**
/	URL of the agent's core user interface.
/<*package*>/	URL of a component of Ego. For example, the ego.core package that implements the core functionality can be found under /core/.
/<*package*>/ia/	URL of exported inter-agent (IA) functions for a package.
/<*package*>/ui/	URL of web browser-renderable HTML pages (UI) for manipulating an agent that are specific to a package.
/<*package*>/ajax/	URL of assorted methods to be used by the UI for manipulating the agent via asynchronous JavaScript calls.
/<*package*>/rest/<*resource*>/	URL of exported RESTful models of a package. For instance, the ego.core package allows its Agent model to be manipulated over REST. So /core/rest/agent/ would be a RESTful endpoint.

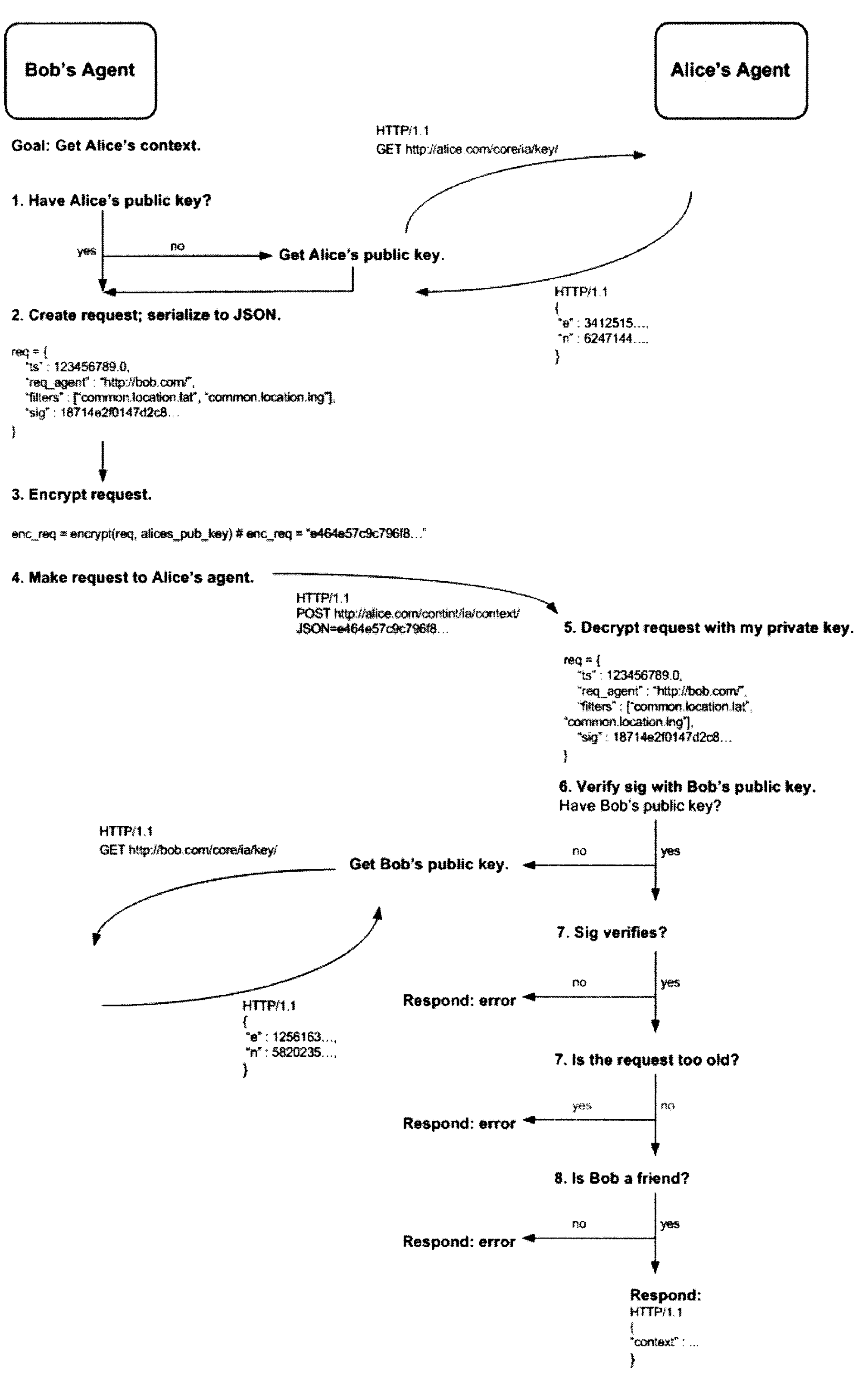

Figure A-1: Agent IA function encryption and authentication workflow.

A.5 API generation

As noted in our discussion of REST (appendix A.2), the protocol's lack of support for auto-generated descriptor formats (such as WSDL) increases difficulty for developers. Developers cannot simply parse a WSDL file in order to list all possible methods; they must read the protocol documentation. We solved this problem in Ego by generating lightweight Python API wrappers from the source documentation.

Ego's documentation is generated by Sphinx [4], the new standard for Python documentation. Sphinx uses the reStructuredText markup language [1] to embed metadata in documentation strings for functions. Sphinx allows extensions to be built to parse custom "directives". We extended Sphinx with three new directives: jsonparams, jsonreturns and httpmethod:

- **jsonparams** - Documents a set of JSON-encoded parameters to the function. The parameters are expected to be contained in the form variable "JSON", regardless of the HTTP request type (POST, GET, etc.) used. Parameters are specified in YAML encoding. YAML was chosen for its clarity and similarity to Python syntax.

- **jsonreturns** - Documents a set of JSON-encoded return values to expect from the function. Like the jsonparams directive, jsonreturns has parameters specified in YAML encoding.

- **httpmethod** - Specifies the HTTP method (GET, POST, PUT, DELETE) to be used when calling this function. Takes one argument which should be one of GET, POST, PUT or DELETE.

An example of this format can be seen in the ContInt plugin's inter-agent protocol specifications:

```
def report(request):
    """
    Reports an identification of this agent.

    .. httpmethod:: POST
    .. jsonparams::
        req_agent               : The URL of the requesting agent
        ts                      : The UTC timestamp of this request in epoch-seconds
        sig                     : The cryptographic hash of this request
```

```
                                using the req_agent's private key
        report: {
            observer            : The URL of the agent that made the observation,
            observed            : The URL of the agent that was observed,
            identifier: {
                pattern         : The pattern used for identification,
                sensor_type     : The type of the sensor that was used,
                agent_url       : The URL of the agent identified by this identifier,
                },
            timestamp           : The time of this report in UTC epoch-seconds,
                }
            meta                : Key-value pairs of metadata about this report. (optional)

    .. jsonreturns::
        resp_agent              : The URL of the responding agent
        errors                  : A list of error strings that occured
        error                   : A boolean indicating whether an error occurred
        ts                      : The UTC timestamp of this request, in epoch-seconds
        sig                     : The cryptographic hash of this response using the
                                  resp_agent's private key
    """
```

The directives allowed us to define special documentation formatting and, as a byproduct, allowed us to generate a Python API around IA functions. The process of calling an IA function on another agent improved dramatically. The old API had a large amount of boilerplate code:

```
import simplejson, urllib, urlparse, time

params = { "req_agent" : "http://my.agent.url.com/",
           "ts" : time.time(),
           "sig" : generate_sig(),
         }
json = simplejson.dumps(params)
params = urllib.urlencode({"JSON" : json})
# POST request
f = urllib.urlopen(urlparse.urljoin(
    "http://ebola.media.mit.edu/camick/",
    "./core/ia/profile/"), params)
response = simplejson.loads(f.read())
f.close()
# Do something with response
```

The new API reduces the boilerplate significantly:

```
import egoapi
my_private_key = "ABCDEF"
egoapi.set_origin("http://my.agent.url.com/", my_private_key)
egoapi.set_target("http://ebola.media.mit.edu/camick/")
```

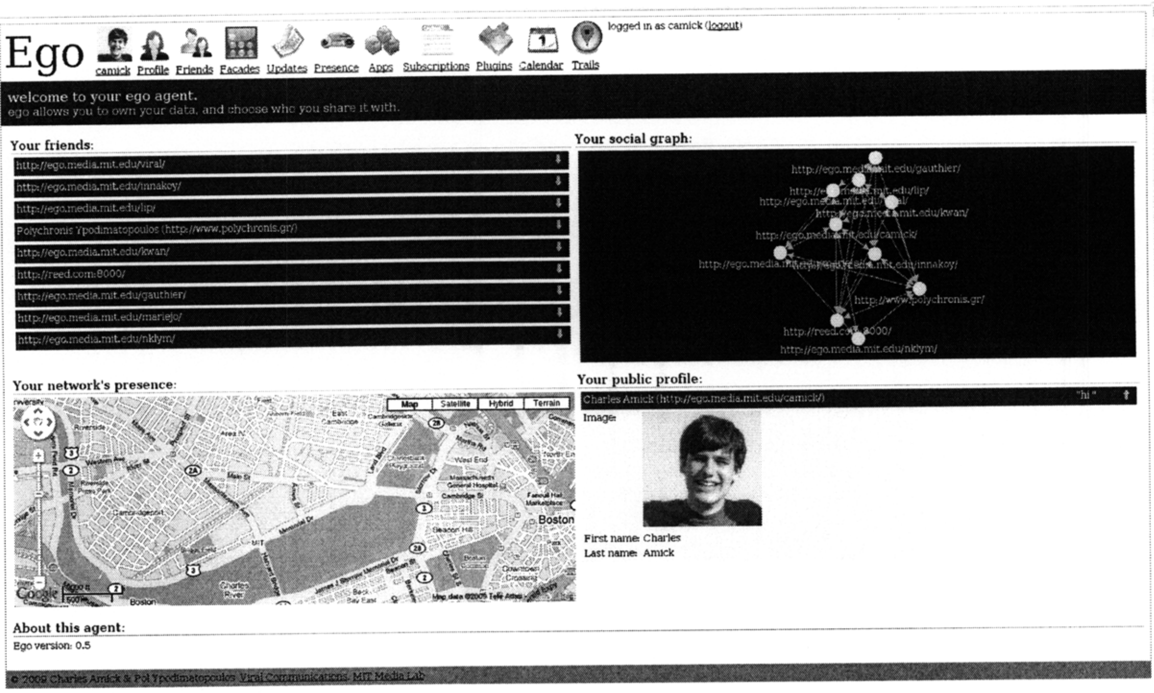

Figure A-2: User interface of the main screen of an agent.

```
response = egoapi.core_ia_profile()
```

Developers can use this new API with little understanding of RESTful APIs and with much less boilerplate code.

A.6 Web-based user interface

Ego provides a web-based user interface from which users can manipulate the data stored in their agent. The interface is dynamic, and is implemented in HTML, CSS and JavaScript. Figure A-2 shows the main screen of an agent; figures A-3 and A-4 show screens oriented around friends and their recent status updates.

ContInt adds user interfaces to manage devices, identifiers, geographical regions of interest (to support region-based events) and a simple visualization of recent contextual data of friends. These four images are shown in Figures A-5, A-6, A-7 and A-8 respectively.

Figure A-3: User interface showing a list of all friends of the agent's owner.

Figure A-4: User interface showing timeline of recent status updates from the friends of the agent's owner.

Ego camick Profile Friends Facades Updates Presence Apps Subscriptions Plugins Calendar Trails logged in as camick (logout)

Presence • Devices • Identifiers • Regions • Wizard

manage your devices.
you can add, edit and remove devices and their sensors here.

Your devices:

n95 Cell phone
Carried with me Sensors Advanced Get settings Get software
Advanced options
Fixed metadata:
Add
common.location.lat
Device options:
Add
wifi.interface

ubuntu Laptop computer
Carried with me Sensors Advanced Get settings Get software
Save changes

Add a new device:

device name, e.g. "My iPhone" Cell phone
Carried with me Sensors Advanced
Save device

© 2009 Charles Amick & Pol Ypodimatopoulos, Viral Communications, MIT Media Lab

Figure A-5: User interface showing device management screen in the ContInt Ego plugin.

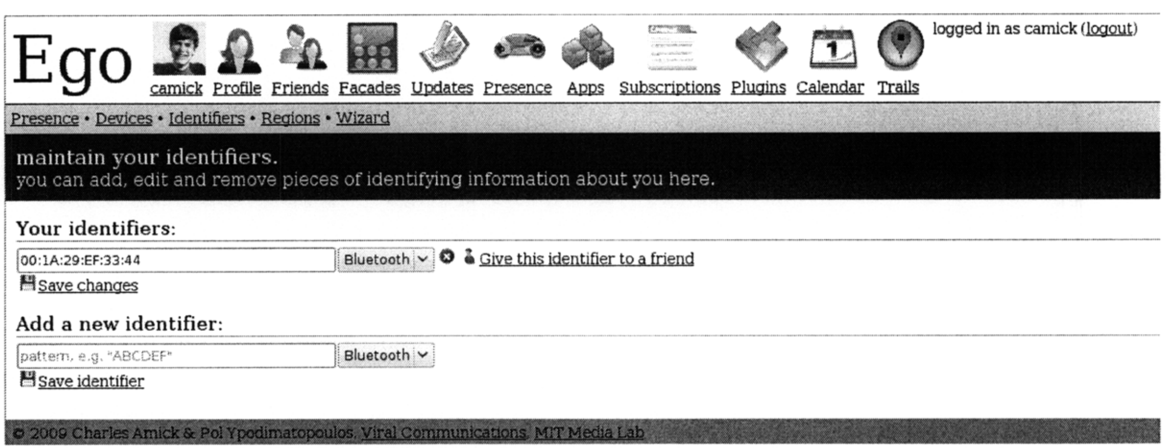

Figure A-6: User interface showing identifier management screen in the ContInt Ego plugin.

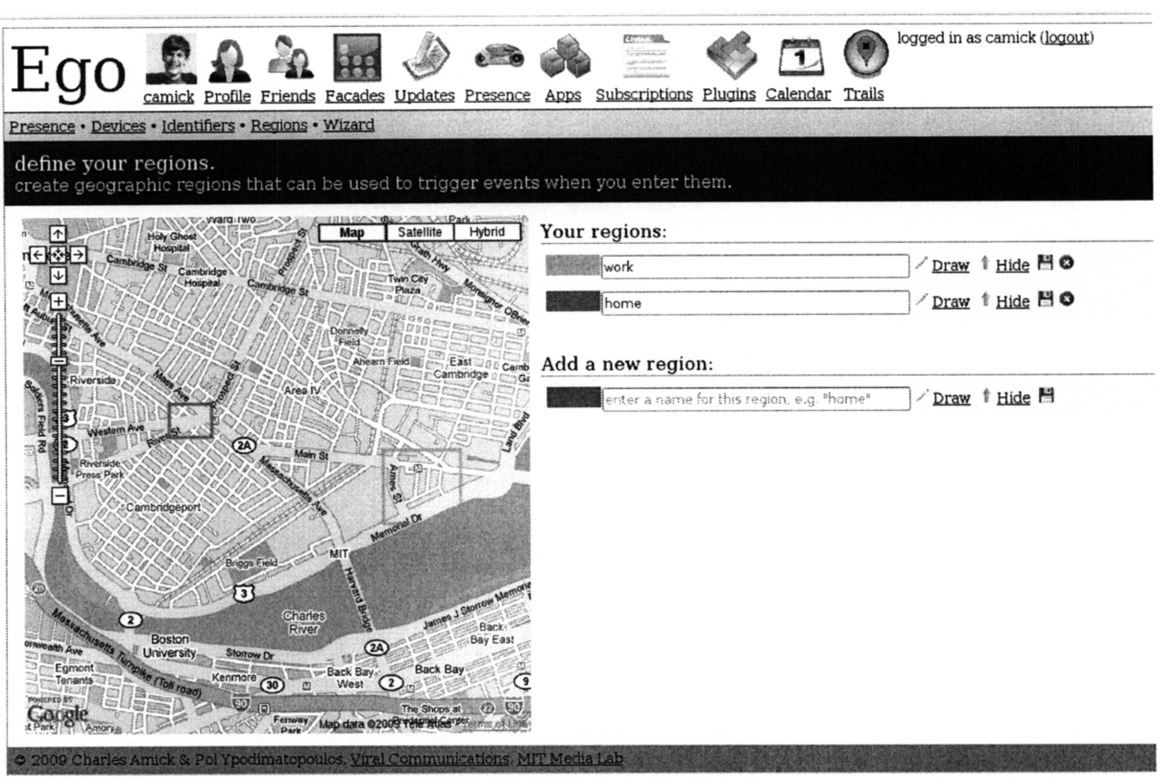

Figure A-7: User interface showing geographic region management in the ContInt Ego plugin.

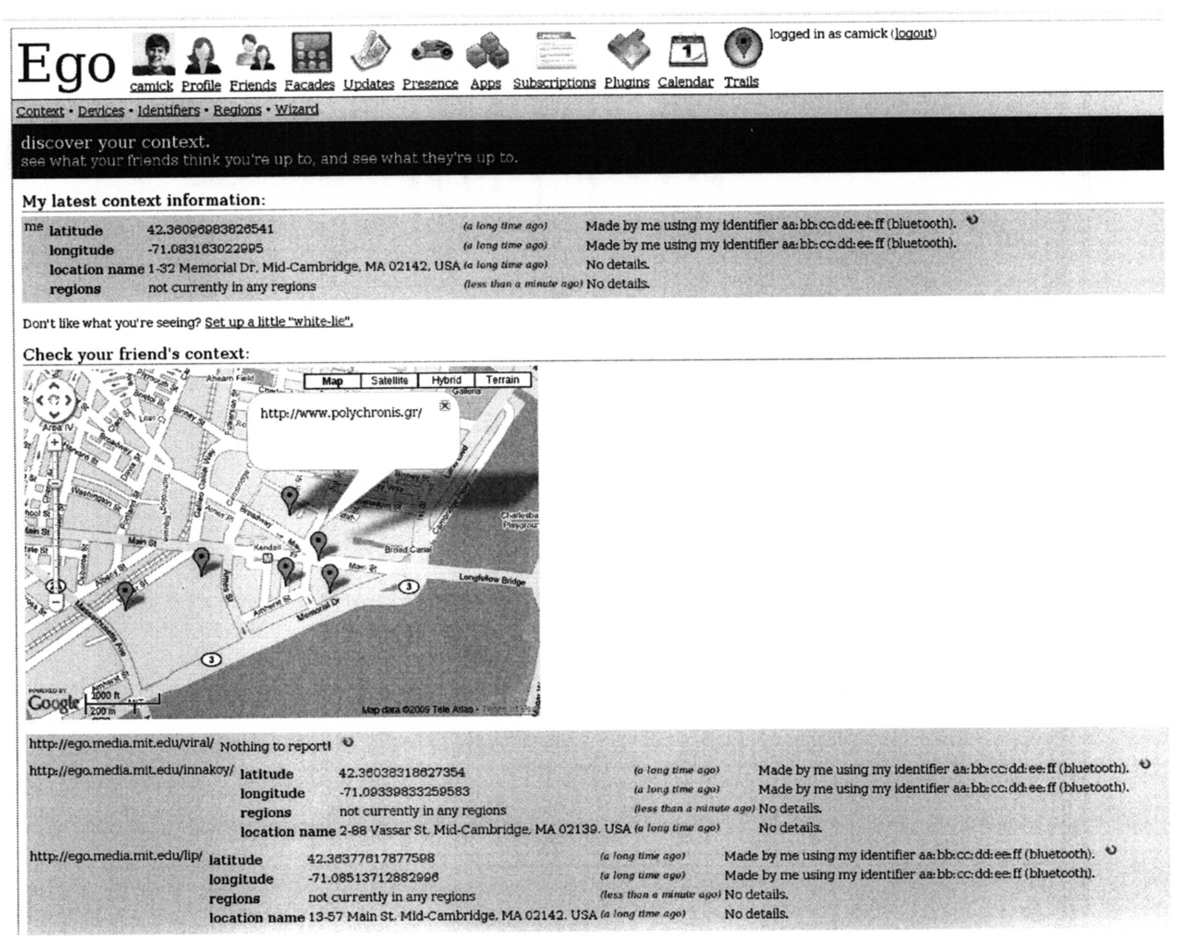

Figure A-8: User interface showing latest contextual data in the ContInt Ego plugin.

A.7 Distributed eventing

A major goal of Ego was to build a publish/subscribe architecture to allow data to flow freely between agents in a social network. The classic observer/observable pattern laid out succinctly by the Gang of Four [21] was applied in order to implement our solution.

We began by implementing a local eventing subsystem that allowed code running on the agent to register function handlers for different event types. When an event is raised, all handlers of the event are called in the order that they were added. The API exposed by the eventing subsystem is simple:

- **register_event_handler** (string **event_type**, function **handler_func**)

  Registers a function to be called when a certain **event_type** is emitted. The **event_type** must exist, or an exception is raised.

- **create_event_type** (string **event_type**)

  Creates a new event type.

- **remove_event_handler** (string **event_type**, function **handler_func**)

  Removes a function that was registered to be called when a certain **event_type** was raised.

- **emit** (string **event_type**)

  Emits the given **event_type**. This causes all handlers of this **event_type** to be called.

Common types of events include "an agent requested your friendship", "new contextual data was reported" and "your friend updated his status." These events are emitted and captured by observers running locally on the agent. Common observers include the logging subsystem that captures and logs all agent-wide events, and plugins. Events allow clean decoupling between different parts of software running on the agent. Plugins are particularly benefited by this separation, as it allows them to tie into existing functionality without having to modify existing code.

Event handler functions have the following method signature:

handler_func (string **event_type**, dict **event_data**, string **src_agent**=None)

The **event_type** is the name of the event that was fired. This is included as some handler functions might be registered to multiple different event types. **event_data** is a dictionary

of arbitrary data associated with the event. For instance, if the event was describing a friendship request, the **event_data** dictionary would contain the requesting agent's URL and a personal message associated with the request. Each event has its own set of data, and it is up to the designer of each **event_type** to document this data clearly. The **src_agent** field contains the URL of the agent that raised this event. It defaults to None; this signifies that the event was raised locally.

The distributed eventing system was built on top of the local eventing subsystem. We added one additional method:

- **register_remote_handler** (string **event_type**, url **remote_url**)

  Registers a remote URL to be notified of events raised on the local agent.

The function wraps the **remote_url** inside a new function that handles the serialization and publishing of the event to registered subscribers. This is done by:

1. Creating a new dictionary, **event**, that contains the raised **event_type** and its **event_data**.

2. Serializing the **event** dictionary into JSON.

3. Posting the serialized **event** dictionary to the **remote_url** with all the necessary fields for secure IA communication (detailed in appendix A.3.)

A.8 Plugin framework

Ego acquires hundreds of megabytes of personal user data; thousands if rich contextual data is maintained. In order to take full advantage of this data for the purposes that Ego is designed for (personal data mining, community sharing), it is extremely important to open up Ego to plugin development. Plugins are small applications that are built on top of the core Ego framework that manipulate its data or add new functionality for data collection and sharing.

A plugin is a self-contained Django application with a few special attributes in its __init__.py file. Creating a plugin is easy for anyone familiar with Django:

1. Create a Django application.

2. Create an __init__.py file in the root of the Django application that contains:

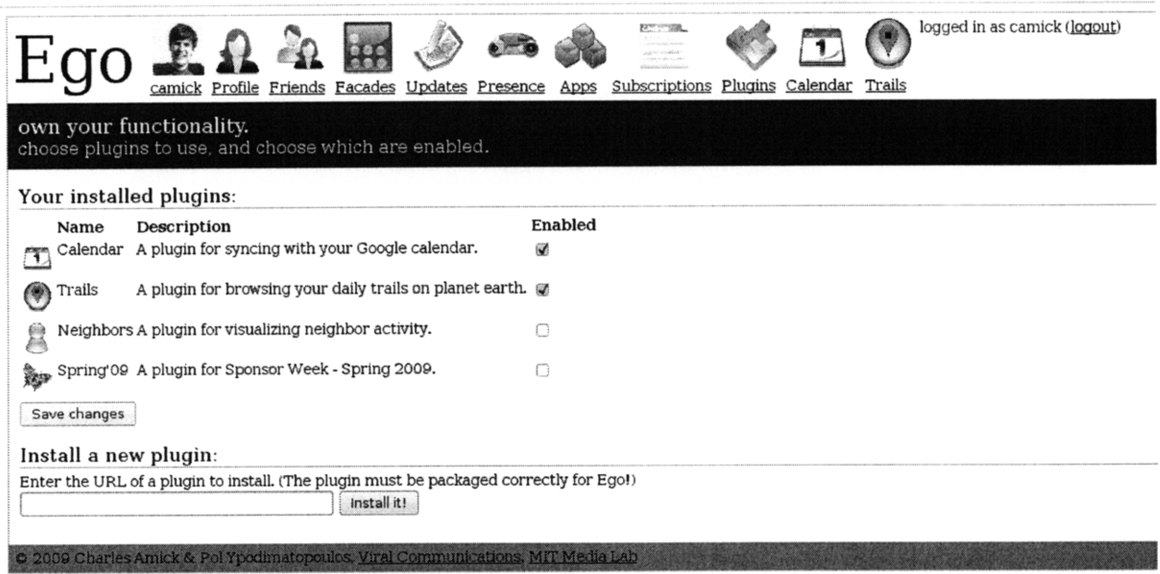

Figure A-9: User interface showing plugin management in Ego.

- **img** (string) - A URL to an image to display in Ego's UI.
- **index** (string) - The name of the view function that is the index of the application. For example, "myplugin.views.index". This is wired to the icon displayed in the Ego website, and will be the first page rendered when users visit the plugin.
- **name** (string) - A name to display in the UI.
- **description** (string) - A short description to be displayed in the plugin management UI.

3. Copy the root directory into Ego's "plugins" subdirectory.

Ego agents periodically check the contents of the plugins subdirectory to see if it has changed. New plugins are loaded dynamically at runtime. Plugins can be enabled and disabled from the plugin control panel in Ego's web UI, shown in Figure A-9.

A.9 Tentacle implementation

There are two different implementations that work together to form ContInt: the ContInt application and the device code that runs on smart devices. The device code, known as the

Tentacle code, polls sensors on the device and reports it back to the owning agent. There are three different Tentacle implementations:

- **Linux** - The Linux implementation is designed for standard Linux distributions. It is implemented in Python. The code has been tested on Ubuntu, Fedora and Maemo (Nokia's Internet tablet Linux distribution). The Linux implementation supports scanning of Bluetooth, capturing images from attached webcams and location sensing with both GPS and WPS (WiFi positioning).
- **PyS60** - The PyS60 implementation is written in Python but targeted at the Symbian S60 [43] platform. S60 is the platform that most of Nokia's cell phones, including the N95, run. The platform is more restricted than Python on a Linux; many standard Python libraries are not available. The PyS60 implementation can determine location through GPS and WPS and can scan for Bluetooth neighbors.
- **Android** - The Android implementation is a Java implementation designed to run on Google's Android G1 cellular phones. It can determine location through GPS and WPS, scan for Bluetooth neighbors, and record accelerometer data.

A.9.1 Workflow

All devices have the same general code structure and implement the same abstract workflow.

1. The device reads a configuration file named "config.json". config.json has the following key-value pair structure:

   ```
   {
       "agent_url" : "http://ego.media.mit.edu/viral/",
       "private_device_id" : "90e35e3776fe9c2a7abfdf3e435920eee1eb1e3e2ef0edec339ebcb1"
   }
   ```

   The **agent_url** stores the URL of the agent to report sensor readings to. The **private_device_id** is the unique id of this reporting device. It is the SHA-224 hexadecimal digest of the string concatenation of the device's database model ID, the time that the device was added to the agent, and a random number between 0 and 10^9. This allows the device to communicate with the agent specified in **agent_url** without

having to retain the user's login credentials on the device. If devices were to identify themselves to their owning agent by simply providing their ID and that ID was nothing more than a function of their creation order, fake devices that spoofed reports to agents could be created.

2. The device connects to the agent specified by **agent_url**, supplying **private_device_id.** The agent responds with the latest configuration settings for the device. (This allows devices to have their configuration settings updated on their agent's web-based user interface and pick up new settings between runs.) The configuration that the device receives has the structure:

   ```
   "config": {
       "main": {
           "agent_url": "http://ego.media.mit.edu/viral/",
           "device_id": 1,
           "sensor_ids": [2]
       },
       "sensor_2": {
           "identifies": true,
           "provides_meta": true,
           "type": "bluetooth",
           "id": 2
       },
       "meta": {
           "common.location.lat": "42.361210",
           "common.location.lng": "-71.087337",
           "common.location.name": "Viral Communications"
       },
       "options": {
           "common.report_period_s" : 40,
       }
   }
   ```

   The configuration always contains 3 sections:

   - **main** - This contains the URL to report to (**agent_url**), the ID of the reporting device (**device_id**), and the IDs of the sensors that are hosted on the device (**sensor_ids**; a list of integers.)
   - **meta** - A collection of key-value pairs of fixed metadata to be injected into every report this device makes (unless overridden by a sensor running on the device): this is fixed contextual data.

- **options** - Contains a set of device-specific options in key-value format.

The above example is from a static sensor that stays in Viral Communications and can only scan for Bluetooth neighbors. Thus the device reports additional metadata including its approximate latitude and longitude, and a location name.

For each **sensor_id** in **main.sensor_ids**, a corresponding entry **"sensor_" + sensor_id** exists. In the above example, there is only one sensor on this device and it is sensor ID 2, so **sensor_2** is a field in the configuration. Each sensor configuration contains four fields: a boolean indicating whether the sensor can identify individuals (Bluetooth and cameras can; a GPS receiver cannot), a boolean indicating whether the sensor provides contextual metadata, the sensor type as a string, and the ID of this sensor in the agent's database.

3. For each sensor in the retrieved configuration, the code instantiates an object of that sensor's type. All constructors take two arguments: a set of device configuration options (as specified in options) and the ID of the sensor.

4. A loop is then entered. The loop sleeps for 30 seconds, or the value of "common.report_period_s" if this key is passed in **options**. When the sleep period has passed, each of the sensors on the device is polled in turn for readings. Each sensor implements a poll function that returns two objects: a list of raw sensor readings that can be used for identification and a list of sensor readings to be used for contextual data. Sensors that do not identify will return empty lists when polled for readings; sensors that do not provide metadata will return empty lists when polled for the metadata.

5. The readings form each device are bundled into a map of key-value pairs and serialized into a JSON object. This object has the form:

```
{
    "device_id" : 1,
    "sensors" : {
        2 : [ {"reading":"abcdefg", "ts":1234567.8},
              {"reading":"hijklmno", "ts":1234568.9} ],
    },
    "meta" : {
        "common.location.lat": "42.361210",
```

```
        "common.location.lng": "-71.087337",
        "common.location.name": "Viral Communications"
    },
}
```


This object is encrypted with the public key of the agent to report to, and is then sent over HTTP to the exposed IA function /contint/p/report/. This is the private reporting API used by devices to report sensings; it is not to be confused with the public reporting API /contint/ia/report/ that allows agents to report sightings to each other.

A.10 Source code

The source code for Ego, ContInt and the Tentacle are available through Subversion [29]. It is not included in this thesis due to its size - at the time of writing, the code is nearly 29,000 lines. The code can be obtained from Subversion:

```
svn co http://enchalla.media.mit.edu/svn/ego/trunk/ego/
```

Details on how to use the code to run an agent are available in tutorials on the Ego website: http://ego.media.mit.edu/.

Bibliography

[1] reStructuredText. http://docutils.sourceforge.net/rst.html, 2009.

[2] MySQL AB and Sun Microsystems. MySQL :: The world's most popular open source database. http://mysql.com/, 2009.

[3] C. Amick and P. Ypodimatopoulos. Ego: Own Your Data! http://ego.media.mit.edu/, 2009.

[4] G. Brandl. Overview - Sphinx v0.6 documentation. http://sphinx.pocoo.org/, 2009.

[5] J. Burke, D. Estrin, M. Hansen, A. Parker, N. Ramanathan, S. Reddy, and M. B. Srivastava. Participatory sensing. In *Workshop on World-Sensor-Web (WSW06): Mobile Device Centric Sensor Networks and Applications*, 2006.

[6] J. Callas, L. Donnerhacke, H. Finney, D. Shaw, and R. Thayer. RFC 4880 - OpenPGP Message Format. http://tools.ietf.org/html/rfc4880, 2007.

[7] CBS. Facebook Gets Guy Dressed As Fairy Bagged By Boss. http://wjz.com/watercooler/facebook.fired.Kevin.2.567620.html, 2007.

[8] One Laptop Per Child. One Laptop Per Child. http://www.laptop.org/, 2009.

[9] SQLite Consortium. SQLite Home Page. http://sqlite.org/, 2009.

[10] Slashdot Contributors. Windows 2000 and Windows NT 4 Source Code Leaks. http://slashdot.org/articles/04/02/12/2114228.shtml?tid=109tid=187, 2004.

[11] Microsoft Corporation. The Official Microsoft IIS Site. http://www.iis.net/, 2009.

[12] M. Dudek. Twopular - Popular trends on Twitter. http://twopular.com/, 2009.

[13] P. Eby. PEP 333 – Python Web Server Gateway Interface v1.0. http://www.python.org/dev/peps/pep-0333/, 2009.

[14] Facebook. API - Facebook Developers Wiki. http://wiki.developers.facebook.com/, 2009.

[15] R. T. Fielding. Architectural Styles and the Design of Network-based Software Architectures. http://www.ics.uci.edu/ fielding/pubs/dissertation/top.htm, 2000. Doctoral disseration, University of California, Irvine.

[16] R. T. Fielding, J. Gettys, J. Mogul, H. Frystyk, L. Masinter, P. Leach, and T. Berners-Lee. Hypertext Transfer Protocol – HTTP/1.1. http://tools.ietf.org/html/rfc2616, 1999.

[17] Apache Software Foundation. The Apache HTTP Server Project. http://httpd.apache.org/, 2009.

[18] Django Software Foundation. Django — The Web framework for perfectionists with deadlines. http://www.djangoproject.com/, 2009.

[19] Python Software Foundation. Python Programming Language – Official Website. http://www.python.org, 2009.

[20] J. Franks, P. Hallam-Baker, J. Hostetler, S. Lawrence, P. Leach, A. Luotonen, and L. Stewart. RFC 2617 - HTTP Authentication: Basic and Digest Access Authentication. http://www.ietf.org/rfc/rfc2617.txt, 1999.

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

[22] C. Gentry. On homomorphic encryption over circuits of arbitrary depth. In *STOC 2009, 41st ACM Symposium on Theory of Computing*, 2009.

[23] Google. Google Latitude. http://www.google.com/latitude, 2009.

[24] PostgreSQL Global Development Group. PostgreSQL: The world's most advanced open source database. http://mysql.com/, 2009.

[25] Viral Communications group. Media Lab Research: Open Spaces. http://www.media.mit.edu/research/1045, 2009.

[26] Viral Communications group. Media Lab Research: The Amulet. http://www.media.mit.edu/research/1045, 2009.

[27] M. Gudgin, M. Hadley, N. Mendelsohn, J. Moreau, H. Nielsen, A. Karmarkar, and Y. Lafon. SOAP Version 1.2 Part 1: Messaging Framework (Second Edition). http://www.w3.org/TR/soap12-part1/, 2007.

[28] J. Hong and J. Landay. An architecture for privacy-sensitive ubiquitous computing. In *MobiSys '04: Proceedings of the 2nd international conference on Mobile systems, applications, and services*, pages 177–189, New York, NY, USA, 2004. ACM.

[29] CollabNet Inc. Subversion open source version control system. http://subversion.tigris.org/, 2009.

[30] Scripting News Inc. XML-RPC Home Page. http://xmlrpc.com/, 2009.

[31] Yahoo! Inc. Fire Eagle. http://fireeagle.yahoo.net, 2009.

[32] Yahoo! Inc. Welcome to Flickr - Photo Sharing. http://www.flickr.com/, 2009.

[33] Yahoo! Inc. Yahoo! Developer Network Home. http://developer.yahoo.com/, 2009.

[34] A. Keränen, J. Ott, and T. Kärkkäinen. The ONE Simulator for DTN Protocol Evaluation. In *SIMUTools '09: Proceeding of the 2nd International Conference on Simulation Tools and Techniques*, New York, NY, USA, 2009. ACM.

[35] A. Lamarca, Y. Chawathe, S. Consolvo, J. Hightower, I. Smith, J. Scott, T. Sohn, J. Howard, J. Hughes, F. Potter, J. Tabert, P. Powledge, G. Borriello, and B. Schilit. Place Lab: Device Positioning Using Radio Beacons in the Wild. In *Proceedings of the Third International Conference on Pervasive Computing*, May 2005.

[36] J. Levine. QDN: Verizon's greed hurts its customers. http://q.queso.com/archives/001903, 2006.

[37] A. Lippmann and D. P. Reed. Viral Communications, 2003.

[38] O. Malik. iPhone is boosting demand for location-based services. http://gigaom.com/2009/04/27/iphone-is-boosting-demand-for-location-based-services/, 2009.

[39] M. McCahill and C. Norris. CCTV in London, 2004.

[40] Memcached. Memcached. http://www.danga.com/memcached/, 2009.

[41] E. Miluzzo, N. Lane, S. Eisenman, and A. Campbell. CenceMe: Injecting Sensing Presence into Social Networking Applications. http://dx.doi.org/10.1007/978-3-540-75696-5_1, 2007.

[42] Fox News. Patriots Cheerleader Fired After Facebook Swastika Photo. http://www.momlogic.com/2008/11/patriots_cheerleader_fired_aft.php, 2008.

[43] Nokia. S60 Home. http://www.s60.com/, 2009.

[44] NTP. ntp.org: Home of the Network Time Protocol. http://www.ntp.org/, 2009.

[45] J. Owyang. Social Networks Site Usage: Visitors, Members, Page Views and Engagement by the Numbers in 2008. http://www.web-strategist.com/blog/2008/11/19/social-networks-site-usage-visitors-members-page-views-and-engagement-by-the-numbers-in-2008/, 2008.

[46] Gilby Productions. TinyURL.com - shorten that long URL into a tiny URL. http://tinyurl.com/, 2009.

[47] Pylons. PylonsHQ - Home. http://www.pylonshq.com/, 2009.

[48] M. Raento. Exploring Privacy for Ubiquitous Computing, 2006.

[49] D. P. Reed. The Third Cloud: New social contexts for safety, 2008.

[50] E. Rescorla. RFC 2818 - HTTP Over TLS. http://www.ietf.org/rfc/rfc2818.txt, 2000.

[51] B. Schilit, A. LaMarca, G. Borriello, W. Griswold, D. McDonald, E. Lazowska, A. Balachandran, J. Hong, and V. Iverson. Challenge: ubiquitous location-aware computing and the place lab initiative. In *WMASH '03: Proceedings of the 1st ACM international workshop on Wireless mobile applications and services on WLAN hotspots*, pages 29–35, New York, NY, USA, 2003. ACM.

[52] A. Semuels. Smartphones now have one-third of market share globally. http://latimesblogs.latimes.com/technology/2009/03/android-smartph.html, 2009.

[53] Bluetooth SIG. Specification of the Bluetooth System Core, Version 1.0B. http://www.bluetooth.com/link/spec/bluetooth$_e$.*pdf*, 1999.

[54] Inc. Skyhook Wireless. Skyhook Wireless. http://www.skyhook.com/, 2009.

[55] T. Sticka. Your Twitter status, in photos – Portwiture. http://portwiture.com/, 2009.

[56] A. Swartz. (web.py). http://webpy.org/, 2009.

[57] K. Tang, P. Keyani, J. Fogarty, and J. Hong. Putting people in their place: an anonymous and privacy-sensitive approach to collecting sensed data in location-based applications. In *CHI '06: Proceedings of the SIGCHI conference on Human Factors in computing systems*, pages 93–102, New York, NY, USA, 2006. ACM.

[58] Twitter. Twitter: What are you doing? http://www.twitter.com/, 2009.

[59] D. Weitzner, H. Abelson, T. Berners-Lee, J. Feigenbaum, J. Hendler, and G. Sussman. Information accountability. *Commun. ACM*, 51(6):82–87, 2008.

[60] Wikipedia. Bluetooth — Wikipedia, The Free Encyclopedia. http://en.wikipedia.org/w/index.php?title=Bluetootholdid=279357721, 2009. [Online; accessed 24-March-2009].

[61] Wikipedia. RSA — Wikipedia, The Free Encyclopedia. http://en.wikipedia.org/w/index.php?title=RSAoldid=279227010, 2009. [Online; accessed 25-March-2009].

[62] WSGI.org. Servers - WSGI Wiki. http://www.wsgi.org/wsgi/Servers, 2009.

[63] P. Ypodimatopoulos. Cerebro: Forming Parallel Internets and Enabling Ultra-Local Economies, 2008.

[64] N. Zhang and C. Todd. A Privacy Agent In Context Aware Ubiquitous Computing Environments, 2006.