Towards a Reliable, Wide-Area Infrastructure for Context-Based Self-management of Communications

Graeme Stevenson*, Paddy Nixon*, and Simon Dobson

Systems Research Group,
School of Computer Science and Informatics,
UCD Dublin, Ireland
`graeme.stevenson@ucd.ie`

Abstract. In this paper we describe *ConStruct*, a distributed, context-aggregation based service infrastructure which supports the development of context-aware applications. ConStruct operates by automatically generating and maintaining directed context-processing graphs which connect applications to the sources of data they require at a relevant level of abstraction. The infrastructure also supports the dynamic creation of context processing elements to bridge gaps between available and requested information. ConStruct provides a reliable, scalable infrastructure; focused on self-maintenance in order to alleviate developer workload. We describe the infrastructure design and implementation, the associated programming model, and our planned extensions to the infrastructure.

1 Introduction

A defining trait of pervasive computing environments is the presence of large numbers of sensors, embedded into the physical surroundings, which provide information on a variety of characteristics of the environment in which they operate. This *context* information can be utilised by applications to modify their behaviour in response to changes in their execution environment, or to convey such changes to users.

The information required by an application is rarely at the same level of abstraction as that provided by individual data sources. For example, a sensor may be able to indicate that a person has been detected in a room, but an application may be interested in the current occupancy of the room, or whether a meeting is currently taking place in the room. In order to obtain such information, an application will frequently have to aggregate data from multiple sources. Such sources may differ in many respects. For example, the level of accuracy they provide, and the data formats and communication protocols they use. As a result, application developers are required to spend the majority of their time on the details

The original version of this chapter was revised: The copyright line was incorrect. This has been corrected. The Erratum to this chapter is available at DOI: 10.1007/978-3-540-32993-0_29

* This work was undertaken while Stevenson and Nixon were at The University of Strathclyde, Glasgow, Scotland.

of obtaining the information they require, rather than on their primary goal of defining the behavioural changes in the applications that use the information.

The aim of our research is to provide a service infrastructure which will carry out the task of obtaining and processing sensor data from a variety of disparate sensor technologies, and deliver it to applications at the level of abstraction they desire. This allows developers to focus on the task of specifying application behaviour with respect to that information.

There are four challenges in the areas of *flexibility*, *maintainability*, *scalability* and *inter-operability* that must be met. Firstly, developers cannot anticipate at development time the physical sources of data that will be available to their applications. Mechanisms must therefore be provided which are flexible enough to support runtime binding between an application and viable data sources. Infrastructure support must also be provided for automatically locating potential data sources, and for bridging the gap between available information and required information using data-aggregation techniques. Secondly, as sensors and data-processing components may be prone to failure, the infrastructure should automatically detect and repair faults wherever possible. Thirdly, an infrastructure should scale to provide support for large numbers of devices, sensors and applications. Finally, as sensors and applications may be deployed on a wide range of heterogeneous devices, standard data formats and communication protocols should be used to provide independence from hardware, operating system, and programming language. Detailed analysis of these requirements can be found in [1].

We have designed and implemented ConStruct, a middleware infrastructure for context-aggregation, with the goal of meeting these challenges. ConStruct draws from several concepts of state-of-the-art context processing research (see Section 3) and extends from this by introducing the automatic synthesis of data sources to bridge gaps between available and requested context information, and by providing mechanisms to reuse components across concurrently executing applications with different data requirements. Work on ConStruct is still in progress, with several strands of research planned to move towards the provision of context in an autonomic computing setting.

The rest of the paper is organised as follows. Section 2 presents an overview of ConStruct. Section 3 presents a discussion of related work. Section 4 discusses Context Entities, the building blocks of the infrastructure. Sections 5 and 6 describe how Context Entities are dynamically composed to provide answers to application queries. In Section 7 we discuss application mobility and communication between multiple instances of the infrastructure services. Section 8 describes the programming model used for developing Context Entities and Applications. We discuss the work still required to meet the goal of context provision in an autonomic computing environment in Section 9, and conclude with a summary in Section 10.

2 Overview of ConStruct

ConStruct is comprised of a number of execution environments called *Ranges*, which self-organise to form a partially connected overlay network referred to as

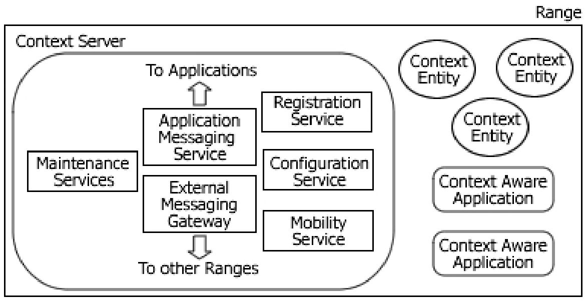

Fig. 1. The set of components which make up ConStruct

the *ConStruct-NET*. Each Range is functionally equivalent and contains a set of services that are used for the management of *Context Entities*, independent units of execution which provide and process context information, and *Context-Aware Applications*, which use the Range services to request and consume the context information produced by entities. Any entity or application which utilises the services provided by a Range is referred to as being a part of that Range. The infrastructure places no restrictions on the physical placement of Range components within the network.

The ConStruct-NET is formed using a self-organising, self-repairing peer-to-peer protocol [2], and provides functionality for dealing with applications which may move between Ranges during their lifetime, and for managing the interactions required to obtain context information from data sources in remote Ranges.

The services provided by a Range are grouped together to form the Context Server. There are six services in total, as shown in Figure 1.

When an entity starts, it sends a request to the Registration Service, advertising the type of information it supplies. This information is used by the Configuration Service, to compose and instantiate graphs of Context Entities, called configurations, which are capable of answering application queries. The External Messaging Gateway is used in this process to obtain context from other Ranges via the ConStruct-NET, whilst the Maintenance Services monitor the status of all the entities and applications in a Range, performing repairs to configurations as required. The Mobility Service is responsible for supporting applications relocating to other Ranges. Finally, the Application Messaging Service provides an additional mechanism for message based entity to application communication outwith the confines of a configuration.

The current implementation of ConStruct is built using the Java Message Service (JMS) [3], a standard, asynchronous messaging API, which supports communication between loosely coupled, distributed components.

3 Related Work

Whilst a lot of work has been undertaken in the field of context delivery over the last decade, the projects closest in spirit to our own are those that provide support for context aggregation. [4] introduces the Contextor, an extension of a Context Widget [5]. Contextors can be composed, and recomposed, into colonies, typed functional units which perform data-aggregation. iQueue [6], provides similar support by automatically combining composers, data aggregation functions written using iQL [7], a purpose built specification language. The iQueue runtime attempts to resolve queries by selecting appropriate data sources using application provided criteria. Finally, Solar [8] allows applications to compose operators using operator-graphs which are instantiated at runtime using available resources. Applications may also specify policies defining how to discard or summarize data flows wherever buffers overflow. Runtime support is provided for load balancing operators across execution environments (planets), for restarting failed operators, and for client mobility.

We note that although existing infrastructures have looked at the problem of automatically generating context-processing graphs (iQueue, Contextors), and context processing across distributed environments (Solar), no project has yet looked at the combination of these elements in tandem with the runtime synthesis of context-processing elements to bridge the gap between available and requested information when only approximate matches are available. This is one of the features of ConStruct.

4 Context Entities

A Context Entity (analogous to a Contextor [4], or Operator [9]), is a lightweight software component which represents either a source of data or a function which operates on the data produced, or computed, by other Context Entities. Each entity has its own thread of execution, and may consume and publish events, which represent context information. This section describes Context Entity meta-data, entity architecture, and the different types of entity supported by the infrastructure.

4.1 Entity Profiles and Naming

It is impractical to require application developers to identify the physical sources for the information they require at development time. Not only would this be a time consuming process for anything more than a trivial application, it would also lead to the development of applications which were inflexible in the face of device failures and changes in the resource pool - two prominent characteristics of their operating environment. To overcome this challenge we require that data sources be identified by the properties of the information they supply rather than by their network location or unique identifier [10].

In order to achieve this, each Context Entity is associated with a profile - XML formatted meta-data which describes the properties of the information supplied by the entity. An entity profile consists of four parts: a classification (see Section 4.3), a location, a description of the output generated by the entity, and descriptions of any inputs the entity requires.

The location parameter describes the logical placement of the Context Entity in the network (based on the Intentional Naming System [11]). For example, an entity representing a coffee machine in room 10 on level 11 of the Livingstone Tower would have the location [LivingstoneTower/L11/R1110].

The description of the output supplied by the entity consists of two parts: a reference to an ontology which describes the format of the events published by the entity, and a set of attribute-value pairs which describe the static properties of the events published by that entity. The property names correspond to those outlined in the ontology.

The combination of entity location and output event description is used to identify resources within the network. This is similar in spirit to Solar [9] and iQueue [6]. It is this format which entities use to express any input requirements they may have.

There is an ongoing effort in the research community towards developing ontologies which can describe the data supplied by a multitude of diverse data sources (e.g., [12]). We assume the existence of such ontologies, although their provision is outside the scope of this work.

We are currently looking at ways in which we can improve the expressiveness of our context specifications. There are two extensions of particular interest. The first allows for the input requirements of a Context Entity to be derived from the output required of it. For example, an entity which will compute the distance in metres between two people (specified at runtime) given their GPS coordinates. The second extension allows Context Entities to define the properties of one input requirement based on the runtime output of another. For example, an entity that provides a list of all the occupants collocated with a given person. This entity requires two inputs: the location of the person we are interested in, and events that describe the detection of people within that location. In order to correctly set up the latter input, we must first have access to the data from the former.

4.2 Entity Architecture

Context Entities consist of three main parts: a control channel, an event channel, and a functional core. Context Entities can send messages to the control channels

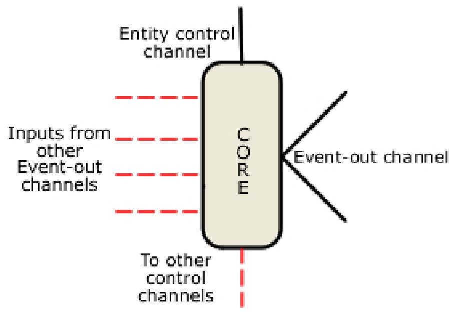
Entity control channel
Inputs from other Event-out channels
C O R E
Event-out channel
To other control channels

Fig. 2. The architecture of a Context Entity

belonging to other entities (or infrastructure services), and may also receive events from the event channels belonging to other entities. This is illustrated in Figure 2.

The functional core of a Context Entity defines how the value of its output events are calculated from its input events, while the control channel of an entity may receive events from the infrastructure services in order to check its status, or from other Context Entities asking it to calculate a new value. These functions discussed in more detail later.

In our current implementation, the control channel corresponds to a JMS Queue - which has one-to-one delivery semantics, whilst the output event channel corresponds to a JMS Topic - which has one-to-many delivery semantics.

4.3 Entity Classification

Context Entities may use data from a wide variety of sources to perform a number of different computations. Influenced by the work in [4], we provide support for seven different flavours of Context Entity. A *source* represents any physical or computational component from which data originates (e.g., a door sensor, or an entity which delivers user preferences). A *fusioner* obtains input from multiple entities which supply events of the same type (X), and outputs events (also of type X) whose quality has improved over that of the input events (e.g., a more accurate estimation of the location of a person based on events produced by RFID and IR sensors). An *aggregator* outputs an event of arbitrary type based on one or more input events, also of arbitrary type (e.g., detecting the activity taking place in a room based on the time, the identity of the people in that room, and the associated noise level). A *transformer* takes an input event of type X and recasts the data into another format without altering its level of abstraction. The output event may be of the same type (e.g., converting a temperature reading from Celsius to Fahrenheit) or a different type (converting data from one event ontology to another). A *generaliser* takes in, and outputs data of type X, where the output data is at a lower level of granularity than that of the input data. We envision the generaliser being used to implement privacy policies, where users may wish to restrict the accuracy of any personal data which is made available to other users (e.g., reducing the accuracy of a location measurement from a room name to a building name). A *filter* takes a single input of type X and outputs a subset of its input events based on some criteria (e.g., to filter out location events about a specific person from a general location service). Finally, a *merger* takes in multiple inputs of type X and outputs each event received without alteration. The purpose of the merger is to aid reuse of event streams and operators (see Section 6.2).

5 Configuration Model

As we described earlier, ConStruct uses the functionality provided by Context Entities to generate answers to queries submitted by Context-Aware

Applications. This is achieved by connecting Context Entities together to form directed, acyclic graphs which produce the required context information as a result. We call these graphs *configurations*. This section describes the architectural style used as the basis for configurations, and describes the interaction model which controls communication between Context Entities.

5.1 Architecture of a Configuration

The architectural style we use for configurations is based upon *Chiron-2* (C2) [13], which was originally devised to support component reuse in GUI based systems. The style consists of a number of components (Context Entities), which are connected together to form a hierarchy using message routing devices (control and event channels). The key property of this style is that components are only aware of other components which reside directly above it in the hierarchy, and have no knowledge of those components which reside lower down. The C2 style supports two forms of communication: notifications, which are passed down the architecture, and requests, which are passed up the architecture. In our case notifications (events) are passed using event channels, while requests are communicated using control channels. Applications represent the lowest level of the hierarchy and form the sinks of the graph.

5.2 Entity Interaction

The interaction model used by ConStruct supports both *active* and *passive* Context Entities. Active entities are characterised by the fact they automatically publish new context information when it becomes available, while passive entities wait until data is requested from them before supplying it. In order to accommodate both types of Context Entity, we use the following interaction model, based on [6]:

– When an Context Entity receives an event from one of its input sources, it will send an event-request message to the entities which lie in the level of the hierarchy directly above it (with the exception of the entity which sent the original event). Once it has received a new event from each of these sources it will calculate and publish a new value.
– When a Context Entity receives an event-request message from an entity (or application) in the level of the hierarchy directly below it, it will send an event-request message to each of the entities which lie in the level of the hierarchy directly above it. Once new values have been returned, the entity will calculate and publish a new value.

We wish to extend our configuration model to provide support for cyclic graphs. This would allow us to support applications and services which employ feedback techniques. We would also like to support the provision of services where entities require to coordinate their efforts with their peers (such as the traffic monitoring/route planning application described in [10]).

6 Query Resolution

The Configuration Service employs *Automatic Path Creation* (APC) techniques in order to generate configurations that are capable of satisfying application queries. This section describes three aspects of this process: the APC mechanism itself, the techniques implemented to reuse existing configurations and entities where possible, and the process of maintaining configurations during their lifetime.

6.1 Query Processing

Restricting, for now, discussion of the resolution process to a single Range, the process carried out by the Configuration Service upon receipt of a query is as follows:

1. First, the Configuration Service searches for Context Entities which match the desired location and output event type requested by the application.
2. The attribute-value pairs describing the output supplied by each candidate entity are then compared to the application's requirements. Entities are classified into one of four categories: *no match*, *partial match*, *exact match*, and *over match*. The no match category contains entities which have conflicting attribute-value pairs to that of the application request. The partial match category contains entities who's attribute-value pairs are a subset of those required by the application. The exact match category includes entities who's attribute-value pairs have exactly one-to-one correspondence with the application request. Finally, the over match category contains entities who's attributes form a superset of those required by the application.
3. If any exact match category contains at least one entity, the next step is to examine each of their input requirements (if any) in turn, and determine if they can be satisfied (using this procedure). This is a recursive process which continues until physical sources of data are found (i.e., Source entities). If there is a choice to be made among multiple entities, the one with the classification that provides the higher quality of data is chosen (e.g., Fusioner > Source > Aggregator).
4. If there are no exact matches, the next step is to examine the input requirements for any partial matches in a similar manner. If a complete configuration can be formed, a filter is automatically generated and configured to bridge the gap between the output of the configuration and the requirement of the application.
5. Should the previous two groups fail to yield a positive result, the final option is to evaluate the group of entities in the over match category. The results of all successfully evaluated configurations can then be merged together to provide the application with the best possible match available.

If the above procedure succeeds in generating a configuration, the Configuration Service sends messages to each entity involved, detailing the identity of the event streams that each should subscribe to. On completion, the Configuration

Service then sends a message to the application informing it about the identity of the event channel representing the end point of the configuration, to which the application will subscribe.

6.2 Reuse of Event Streams and Context Entities

In the process of generating a configuration, the Configuration Service will try to reuse active event streams and Context Entities (i.e., those which are part of an existing configuration) wherever possible. In the case of event streams, this is a straightforward process. If the output of an active entity is found to satisfy either the application query itself, or one of the inputs required by a entity within the new configuration, the Configuration Service will utilise the existing event stream, thus satisfying that particular branch of the configuration completely.

In the case of Context Entities, the process is slightly more involved. When we talk about reusing a Context Entity, we refer to the functionality of the entity, rather than the role it plays in an existing configuration. For example, an entity that converts context information from one ontology to another can perform the same role in multiple configurations which require information from different sources. The process of reusing an entity in multiple configurations involves merging the desired event streams, tagging each stream with an identifier, passing the event stream through the reusable entities, and finally filtering out the original event streams. This is illustrated in Figure 3.

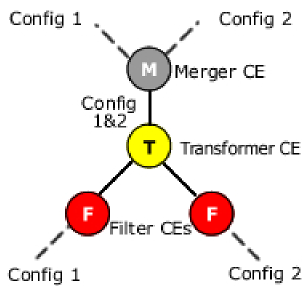

Fig. 3. Example showing the reuse of a transformer entity across two configurations

6.3 Runtime Maintenance

Pervasive computing environments are considered to be dynamic with respect to the resources available within them at any one time. Another tenet of such environments is that the failure of computational devices should be treated as commonplace. To deal with these features, ConStruct provides a suite of maintenance services that: monitors Context Entities and Applications for failure; performs repairs to configurations where possible; and re-evaluates configurations when new resources become available.

Application and Context Entity monitoring takes the form of periodic passing of ping/pong style messages. If the control channel of an application/entity has been closed or if a response has not been received within a set number of iterations the application/entity is assumed to have failed. In the case of an application failure, any configurations to which they were the sole subscriber can be removed, and the involved entities told to deactivate. In the case of an entity failure, the Configuration Service will be used to try and repair the branches of any configurations which utilise the entity. If a configuration can be repaired successfully and the end point of the configuration is unchanged there is no need to inform the application. Otherwise, the application is told to change their subscription, or that their requirements can no longer be satisfied.

Periodic re-evaluations of queries are performed in order to take advantage of additions to the resource pool. Should a preferable configuration be found to one already in use, the affected branches of a configuration are altered, old branches deactivated and applications informed as above if necessary. We are currently working on providing support for runtime configuration adaption based on changing Quality of Service parameters (e.g., accuracy, confidence, error, and bandwidth). Although this information has always been available (should an entity choose to provide it), its interpretation was previously left to data consumers. As different data types have different QoS parameters associated with them, we aim to develop an model which is extensible, allowing us to perform informed analysis of the QoS parameters of new data types as they emerge.

7 ConStruct-NET

The ConStruct-NET facilitates the communication of context information over a wide-area by connecting distributed Ranges. This allows applications (by way of the infrastructure services) access to the context information they require, irrespective of their network location and Range they are part of. This section gives a brief overview of the communication mechanisms employed to form the ConStruct-NET, the extension of the configuration model to communicate with data sources located in remote Ranges, and the infrastructure mechanisms that support application mobility between Ranges.

7.1 Inter-range Communication

The *External Messaging Gateway* (EMG) component of a Range is responsible for initialising (or joining) the ConStruct-NET, and for all communication between remote Ranges and its own. The ConStruct-NET is implemented using *Pastry* [2], which provides the basis for communications, and message routing within peer-to-peer applications. Details on the message routing algorithm employed by Pastry, and the self-organising and self-repairing characteristics of a Pastry network are described in [2].

7.2 Extending the Configuration Model

In order that the context information supplied by entities in remote Ranges can be utilised, we impose two additional requirements on the single Range

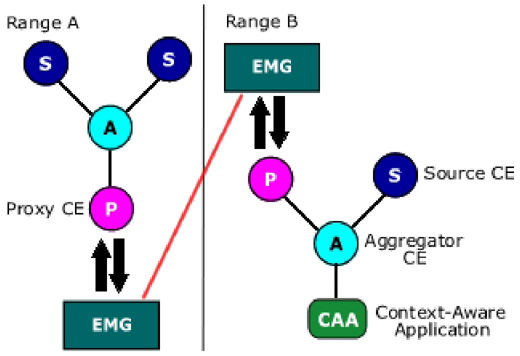

Fig. 4. Example showing the use of a proxy to communicate information across the ConStruct-NET

model described above. Firstly, that the query resolution algorithm incorporates searches across multiple Ranges. Secondly, that the model used for executing configurations is extended to provide support for obtaining information from and sending requests to entities across the ConStruct-NET.

If part (or all) of a configuration cannot be resolved locally, the Configuration Service will route a request through the EMG to the Configuration Services belonging to other Ranges, which will attempt to complete the configuration. Should a remote Configuration Service be able to contribute to the configuration, the process will recurse from that point in a similar manner until the configuration is completed, or the process fails.

In order that that our configuration model remains consistent, we have introduced *proxy* Context Entities, which bridge the gap between Ranges, serving as a local representation of a remote entity. Proxy entities use the EMG to communicate with the entity they represent. This is illustrated in Figure 4.

7.3 Application Mobility

Mobile applications may use the Mobility Service to retain their configurations whilst they relocate to another Range, or during periods where they experience temporary loss of connectivity (e.g., out of range of a Wi-Fi access point). The Mobility Service acts as a proxy between an application and the end point of its configurations. Should message delivery to the application fail, the Mobility Service will cache events on the applications behalf. When an application rejoins the ConStruct-NET (either in the same or a different Range), it uses the infrastructure to route a message to the Mobility Service, asking it to resume message delivery. Should an application fail to reappear within a reasonable time period (set by the administrator of a Range), the assumption is made that the application will not return. At this point the Mobility Service will stop acting on the application's behalf, and the maintenance procedures will perform cleanup operations as normal.

8 Programming Model

ConStruct provides a simple two-step programming model that allows developers to easily create their own Context-Aware Applications and Context Entities.

To create a new Context Entity, the first step is to extend from the entity base class which provides all the functionality required to interact with the infrastructure services. Developers are required only to provide an implementation for the *evaluate()* method, which returns an XML encoded String representing the event produced by that entity. Access to any inputs required by the entity are achieved by calling the *getSource("sourceName")* method. Updated events for these inputs are obtained automatically (see Section 5.2) before the *evaluate()* method is called. The developer has the option of specifying how often the entity should evaluate as part of its constructor. If no value is given, the entity is treated as being passive.

Similarly, the base class from which applications extend only requires developers to provide an implementation for the *eventHook()* method. This is called automatically when an application receives a new event.

The second step in the development process is to write the XML context specification for the entity (its profile) or application (its queries). Profiles include the entity location, input, and output details of a entity as discussed in Section 4.1, whilst the format used for a specifying each application query includes a local name (used as a parameter when the *eventHook()* method is called, signalling which query the event is associated with), location, event type, and associated attribute-value property list. Context specifications are stored in an external text file, and identified to the entity/application through the object's constructor. The process of verifying and using the data provided by the profiles and queries is handled automatically.

We have build several applications using our programming model, including an In/Out Board, a Context-Aware Coffee Break Notifier, and a Smart To-Do List. Details of these can be found in [1].

9 Towards Context Provision in an AC Environment

Up to this point, the focus of our work has been on investigating the necessary abstractions to allow us to decompose high-level services into low-level building blocks, and on the mechanisms to facilitate their communication and reuse. The techniques we use in this process have allowed us to place all the maintenance complexity and communication logic into the software, minimising the effort required of developers to build mobile applications which can source data from any location within the ConStruct-NET.

In addition to the ongoing work we have described throughout this paper, there are several issues which we must address before we reach the stage of providing a context delivery mechanism which is suitable for an autonomous networking environment.

Whilst the infrastructure has self-organising and self-healing properties at the Range level, both in terms of the Pastry network protocol used to form the

ConStruct-NET and the fact that it provides automatic creation and maintenance of configurations within a Range, at the macro level we have a reliance on centralised services. This single point of failure, whilst effective at lessening the processing burden on individual devices, is a fair criticism of our work from an autonomic communications perspective.

We aim look at the feasibility of decentralising our protocols. This raises a number of issues, such as: the efficiency of the discovery protocol; the time required to construct a configuration; memory footprint; CPU load, which will be of critical importance for battery powered devices; and preserving the facility to source data from remote locations. Another key issue involves the synthesis and reuse of data sources - where we currently use the infrastructure services to do this work, another approach will be required. The concept of a domain, as we have with a Range is useful, and retaining this concept when decentralising our protocol is something to consider.

Finally, security of data is also an important issue - primarily in terms of access control, although encryption may be a requirement in some cases. Providing access control mechanisms for dealing with context is a complex issue. Challenges include the need to determine ownership of the data; to resolve conflicting privacy preferences (between users and/or administrative domains), and to provide mechanisms for permitting access to information at different levels of granularity (e.g., granting access to your location information at room, building, or city level depending on the identity of the interested party).

10 Summary

In this paper we have described ConStruct, a service infrastructure designed to enable the collection and aggregation of context information from a myriad of distributed data sources, and the distribution of that information to the applications that require it at an appropriate level of abstraction. We detailed the mechanisms which we use to allow runtime synthesis of new data sources to bridge processing gaps, and the techniques we use to support the reuse of processing elements across multiple configurations. We also described how ConStruct facilitates context processing and dissemination over a wide-area using multiple deployments of the infrastructure services. We concluded this paper by discussing some of the issues that we still need to address in order to apply our technology to the autonomic computing domain.

References

1. Graeme Stevenson. A Service Infrastructure for Change-Tolerant Context-Aware Applications. Master's thesis, University of Strathclyde, Glasgow, Scotland, 2005.
2. Antony Rowstron and Peter Druschel. Pastry: Scalable, Decentralized Object Location, and Routing for Large-Scale Peer-to-Peer Systems. *Lecture Notes in Computer Science*, 2218:329–350, 2001.
3. R. Monson-Haefel and D. Chappell. *Java Message Service.* O'Reilley & Associates, December 2000.

4. Jolle Coutaz and Gatan Rey. Foundations for a Theory of Contextors. In *Computer-Aided Design of User Interface (CADUI02)*, 2002.
5. Anind Dey. *Providing Architectural Support for Building Context-Aware Applications*. PhD thesis, Providing Architectural Support for Building Context-Aware Applications, November 2000.
6. Norman H. Cohen, Apratim Purakayastha, Luke Wong, and Danny L. Yeh. iQueue: A Pervasive Data Composition Framework. In *Third International Conference on Mobile Data Management (MDM'02)*, pages 146–153, Singapore, January 2002.
7. Norman H. Cohen, Hui Lei, Paul Castro, John S. Davis II, and Apratim Purakayastha. Composing Pervasive Data Using iQL. In *Proceedings of the Fourth IEEE Workshop on Mobile Computing Systems and Applications*, page 94. IEEE Computer Society, 2002.
8. Guanling Chen and David Kotz. Application-controlled loss-tolerant data dissemination. Technical Report TR2004-488, Dartmouth College, Computer Science, Hanover, NH, February 2004.
9. Guanling Chen and David Kotz. Context Aggregation and Dissemination in Ubiquitous Computing Systems. In *The Fourth IEEE Workshop on Mobile Computing Systems and Applications.*, pages 115–114, Callicoon, New York, June 2002.
10. Norman H. Cohen, Apratim Purakayastha, John Turek, Luke Wong, and Danny Yeh. Challenges in Flexible Aggregation of Pervasive Data, 2001.
11. William Adjie-Winoto, Elliot Schwartz, Hari Balakrishnan, and Jeremy Lilley. The design and implementation of an intentional naming system. In *Symposium on Operating Systems Principles*, pages 186–201, 1999.
12. Harry Chen, Filip Perich, Tim Finin, and Anupam Joshi. SOUPA: Standard Ontology for Ubiquitous and Pervasive Applications. In *First Annual International Conference on Mobile and Ubiquitous Systems: Networking and Services (MobiQuitous'04)*, Boston, Massachussets, USA, August 2004.
13. Richard N. Taylor, Nenad Medvidovic, Kenneth M. Anderson, E. James Whitehead Jr., Jason E. Robbins, Kari A. Nies, Peyman Oreizy, and Deborah L. Dubrow. A Component- and Message-Based Architectural Style for GUI Software. *Software Engineering*, 22(6):390–406, 1996.