Report Number: WUCSE-2004-10

2004-03-15

Context-Sensitive Data Structures Supporting Software Development in Mobile Ad Hoc Networks

Jamie Payton, Gruia-Catalin Roman, and Christine Julien

Context-aware computing, an emerging paradigm in which applications sense and adapt their behavior to changes in their operational environment, is key to de-veloping dependable agent-based software systems for use in the often unpredictable settings of ad hoc net-works. However, designing an application agent which gathers, maintains, and adapts to context can be a diffi-cult undertaking in an open and continuously changing environment, even for a seasoned programmer. Our goal is to simplify the programming task by hiding such issues from the programmer, allowing one to quickly and reliably produce a context-aware application agent for use in large-scale ad hoc... **Read complete abstract on page 2.**

Context-Sensitive Data Structures Supporting Software Development in Mobile Ad Hoc Networks

Jamie Payton, Gruia-Catalin Roman, and Christine Julien

Complete Abstract:

Context-aware computing, an emerging paradigm in which applications sense and adapt their behavior to changes in their operational environment, is key to de-veloping dependable agent-based software systems for use in the often unpredictable settings of ad hoc net-works. However, designing an application agent which gathers, maintains, and adapts to context can be a diffi-cult undertaking in an open and continuously changing environment, even for a seasoned programmer. Our goal is to simplify the programming task by hiding such issues from the programmer, allowing one to quickly and reliably produce a context-aware application agent for use in large-scale ad hoc networks. With this goal in mind, we introduce a novel abstraction called context-sensitive data structures (CSDS). The programmer in-teracts with the CSDS through a familiar program-ming interface, without direct knowledge of the context gathering and maintenance tasks that occur behind the scenes. In this paper, we begin by defining a model of context-sensitive data structures, and we identify key requirements and issues associated with building an infrastructure to support the development of context-sensitive data structures.

2004-10

Context-Sensitive Data Structures Supporting Software Development in Mobile Ad Hoc Networks

Authors: Payton, Jamie; Roman, Gruia-Catalin; Julien, Christine

March 15, 2004

Abstract: Context-aware computing, an emerging paradigm in which applications sense and adapt their behavior to changes in their operational environment, is key to developing dependable agent-based software systems for use in the often unpredictable settings of ad hoc networks. However, designing an application agent which gathers, maintains, and adapts to context can be a difficult undertaking in an open and continuously changing environment, even for a seasoned programmer. Our goal is to simplify the programming task by hiding such issues from the programmer, allowing one to quickly and reliably produce a context-aware application agent for use in large-scale ad hoc networks. With this goal in mind, we introduce a novel abstraction called context-sensitive data structures (CSDS). The programmer interacts with the CSDS through a familiar programming interface, without direct knowledge of the context gathering and maintenance tasks that occur behind the scenes. In this paper, we begin by defining a model of context-sensitive data structures, and we identify key requirements and issues associated with building an infrastructure to support the development of context-sensitive data structures.

Context-Sensitive Data Structures Supporting Software Development in Ad Hoc Mobile Settings

Jamie Payton, Gruia-Catalin Roman, and Christine Julien
Department of Computer Science and Engineering
Washington University in St. Louis
Campus Box 1045, One Brookings Drive
St. Louis, MO 63130-4899, USA
{payton, roman, julien}@wustl.edu

Abstract

Context-aware computing, an emerging paradigm in which applications sense and adapt their behavior to changes in their operational environment, is key to developing dependable agent-based software systems for use in the often unpredictable settings of ad hoc networks. However, designing an application agent which gathers, maintains, and adapts to context can be a difficult undertaking in an open and continuously changing environment, even for a seasoned programmer. Our goal is to simplify the programming task by hiding such issues from the programmer, allowing one to quickly and reliably produce a context-aware application agent for use in large-scale ad hoc networks. With this goal in mind, we introduce a novel abstraction called context-sensitive data structures (CSDS). The programmer interacts with the CSDS through a familiar programming interface, without direct knowledge of the context gathering and maintenance tasks that occur behind the scenes. In this paper, we begin by defining a model of context-sensitive data structures, and we identify key requirements and issues associated with building an infrastructure to support the development of context-sensitive data structures.

1. Introduction

In recent years, communication technology has begun to reflect the dynamic nature of society, with devices becoming increasingly portable and untethered. The widespread use of mobile devices brings about an increased demand for software designed with mobility in mind. In fact, we can expect the number of software systems designed for use in ad hoc networks to experience rapid growth. In such networks, connections are formed opportunistically between devices within wireless communication range. Applications for this environment are likely to come into routine usage in situations such as disaster recovery in which rescue workers must find and treat victims, construction supervision in which a foreman gathers information around a site to gauge progress, etc. These and other applications for ad hoc networks are often composed from several application agents that must operate in open and highly dynamic environments, making it difficult for the programmer to produce reliable and dependable software.

Context-aware computing has been advocated as a solution for managing the programming complexity associated with such development efforts. Context-awareness refers to the ability of a software system to adapt its behavior in response to environmental changes. Typical examples of context-aware systems include location-aware offices (e.g., Active Badge [6] and PARCTAb [12]), context-sensitive tour guides (e.g., Cyberguide [1] and GUIDE [3]), and context-aware note tools (e.g., FieldNote [10]). Constructing such systems is a daunting task, requiring the developer to consider the interaction between the system and a number of possibly heterogeneous sensors to gather and deliver context information.

Several frameworks and infrastructures have been devised to promote efficient, reliable development of context-aware applications by masking the complexity of interacting with heterogeneous sensors, e.g., the Context Toolkit [11] and the Context Fabric [7]. While these support systems simplify interactions with sensors, the programmer must still know the source of data to access and operate on it. In an ad hoc network, the open and dynamic nature of the environment makes it unreasonable to assume advance knowledge of the identities of data sources; application agents for use in such scenarios require a highly decoupled method of data access. Mobile agent middleware systems have been developed that provide decoupled communication in ad hoc networks, including LIME [9], MARS [2], and EgoSpaces [8]. Many of these systems, however, are tied to the tuple space data abstraction.

To provide a more general and flexible method of decoupled data access for application agents operating in an ad hoc network, we propose the concept of context-sensitive data structures as the basis for a new programming methodology. A context-sensitive data structure (CSDS) is determined by and provides access to data available in the context; it is encapsulated as an abstract data type (ADT), which is represented by a class in a programming language such as Java or C++. Like all classes, it provides the programmer with an application programming interface (API) to access and manipulate data. The collection of data items operated on by an instantiation of such a class changes with the content of the ad hoc network. The distributed data items are accessed using the API of the local class instantiation.

The resulting design methodology provides the designer with the flexibility to use familiar and proven programming tools, i.e., ADTs, for context-aware application development. The programming tasks associated with gathering, maintaining, and adapting to context are simplified for the developer, which allows the focus to be shifted to satisfying domain-specific requirements. While implementations of context-sensitive data structures may be useful to the burgeoning community of context-aware application developers, requiring programmers to construct an entire library of these data structures from scratch is impractical. Our goal is to provide a general model and infrastructure to support the gradual development of a library of context-sensitive data structures, which can, in turn, be used to support the context-sensitive data structures programming methodology. In this paper, we lay the conceptual foundation required to support the methodology by defining the context-sensitive data structures model and by exploring the needs of CSDS developers.

The remainder of this paper is organized as follows. Section 2 summarizes the computational model and the notion of context assumed in this paper. A motivating example of a CSDS and its use in developing a context-aware application agent is given in Section 3. Section 4 addresses the key elements required in an infrastructure for supporting the development of context-sensitive versions of traditional data structures and discusses issues with developing protocols for inclusion in the infrastructure. Conclusions appear in Section 5.

2. Context-Sensitive Data Structures Explained

As we embark on an exploration of the context-sensitive data structures model, we should be more specific about the environment in which an application operates. We consider agents as the main computational entities of a system, as well as providers and users of data items. To put it simply, agents are pieces of code that make up an application. Pieces of data provided by an agent are context items; context items have a general representation and can capture a wide range of information that may be important to an application, e.g., sensor readings, location information, etc. Runtime support for agents is provided by devices, which simply serve as containers for agents; we often refer to such devices as hosts. Hosts perform no application execution and do not provide or use data. An agent may migrate between connected hosts. Hosts are connected when they are within wireless communication range, and agents are connected when they reside on the same host or on connected hosts. This definition of agent connectivity is important in our definition of context. In the systems that we consider, each agent has an individual context. An agent's maximal context consists of data items provided by connected agents. When talking about a particular agent's context, we often refer to that agent as the *reference agent*.

Context-sensitive data structures are an appropriate abstraction for accessing and operating on the data available in an agent's context. To allow applications in large scale multi-agent systems to economically manage an expansive context, an agent should be supplied with a pertinent and manageable subset of the maximal context as its tailored context. The context for a particular data structure is specified by the agent, and context-sensitive data structures operate on these tailored definitions of the application context. Thus, a context-sensitive data structure's content is determined by the state of the environment and a specification of context supplied by the application. In terms of our computational model, the content of the CSDS is defined as all data items on agents running on hosts in the logical subnet of the ad hoc network that forms the application context meeting some qualification criteria.

When a context-sensitive data structure is used, the task of managing access to the data elements that are spread across the ad hoc network while maintaing a specific data organization possibly defined by the structure is hidden from the application programmer. Access to the data elements of the CSDS is gained only through operations defined on its ADT. Operations performed on the CSDS can effect a change in the context of others, as can the movement of an agent that may cause it to join or leave someone else's context. As these changes in the state of the environment occur, the content of the context-sensitive data structure is changed appropriately in response. A developer using a context-sensitive data structure can operate on the dynamically changing set of data elements that are distributed throughout the context as if the data were stored in a local, persistent data structure. The management of the data elements within the CSDS is automatically handled in the face of changes without intervention by the application programmer, since the data structure is essentially a reflection of context.

In the remainder of this paper, we investigate the software engineering potential for context-sensitive data structures. First, we offer a concrete example of a context-aware application that can benefit from the use of a particular context-sensitive data structure, the priority queue. We then explore protocols we must provide in the infrastructure to support the development of context-sensitive data structures for use in such applications. In doing so, we seek to demonstrate the feasibility of applying the context-sensitive data structures concept and associated design methodology.

3. Programming with Context-Sensitive Data Structures

The impetus behind the introduction of the context-sensitive data structure design methodology is to reduce development costs in terms of effort and errors, and to make context-aware application development accessible even to novice programmers. Context-sensitive data structures provide a decoupled method of accessing and operating on data in the ad hoc network, one that is simple and natural to the programmer, using the same interface as in static settings. Moreover, the dynamically changing content is managed transparently, reducing the complexity of the environment, and, in turn, the potential for programming errors incurred by interacting with agents in a large-scale and highly dynamic ad hoc network.

To illustrate the utility of context-sensitive data structures and the associated design methodology, consider a disaster recovery scenario in which triage is employed to treat the wounded. Victims are quickly examined to evaluate the seriousness of their injuries and are tagged with devices that emit (via wireless radio or infrared) information about the assigned injury classification, ranging from injuries that need immediate attention to those for which treatment can be postponed. Rescue teams are assigned areas in which they must arrange transport for the most severely injured first and provide as much on-site treatment as possible for these victims until transport is available. The rescue team members use PDAs with wireless communication capabilities to coordinate activites and to obtain and display the status of victims and volunteers. A volunteer is selected by the rescue team member to treat the most seriously wounded victim until transport arrives. A volunteer's assignment may change as the status of injured victims within the context changes. After a rescue crew member arranges on-site treatment for victim, he must arrange for the victim's transport to a hospital. As victims are transported, they are removed from the context of the application. As new victims are discovered and their injuries evaluated, they are added to the context. Figure 3 illustrates this application.

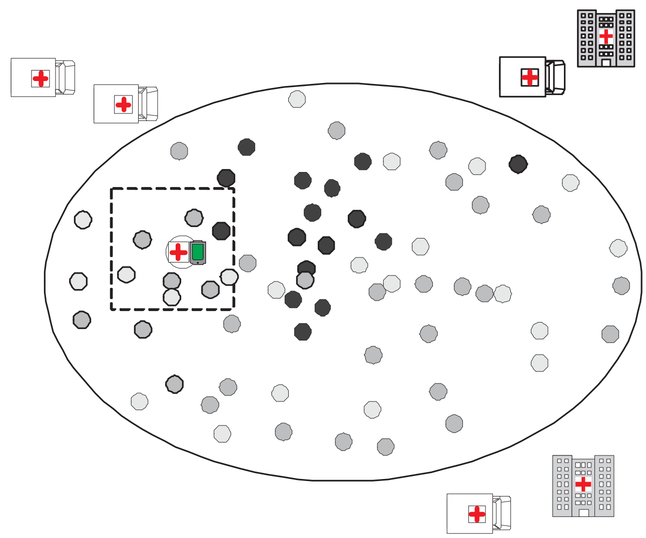

Figure 1. Disaster Recovery Scenario. The disaster site lies within the large oval. A rescue crew member (the encircled cross) uses a PDA that runs an application to assign to the most seriously wounded victims in the designated area (the dashed box) on-site treatement and ambulance transport to a nearby hospital. Victims are shown as circles, with seriousness of injury reflected by darker shading.

Building the application described from scratch can be a significant undertaking. The programmer must include functions to sense the set of neighboring hosts, to send messages to agents on reachable hosts, and to issue queries to obtain data. Query responses must be processed and placed into a traditional, static priority queue. When an operation is requested, the hosts in the network must be queried to ensure operation over a set of data most closely reflecting the current state of the context. The remainder of this section illustrates a CSDS approach to implementing the disaster recovery application and demonstrates how context-sensitive ADTs can be used by application programmers.

When using the CSDS programming methodology, the amount of data processed by the application is reduced, explicit data maintenance by the application programmer is removed, and application development is simplified. A simple application agent for rescue team support could be constructed around the notion of a context-sensitive priority queue. Within this priority queue, the content of the data structure is defined by a context specification that restricts the context to a manageable area of the disaster site, e.g., a one block radius. The data associated with the priority queue reflects an ordering over the injured within that area such that the most seriously injured victim is at the head of the queue. The context, and hence the content of the context-sensitive priority queue, is updated independently of the application's operation on the queue.

Data elements of the priority queue used in the triage application are pieces of injury information emitted from victims' triage tags. The content of a context-sensitive priority queue for a particular rescue crew member's application is determined by the context definition provided. The context definition restricts which elements of the entire operational environment will be included as items in the priority queue. In this case, the content of the priority queue for the crew member's application is the injury information data elements that are located within her assigned area.

We envision our priority queue as having two operations: `getFirst()` and `removeFirst()`. In our application, `getFirst()` is used to access an injury description for the victim in the context with the most severe injury. The injury description includes a unique injury identifier, the injury priority, and the geographical location of the injured person. The `removeFirst()` operation is used to access the injury description of the victim with the highest priority injury and to remove the injury description from the queue.

It may seem that we have omitted operations needed to populate a priority queue. While explicit data insertion operations may be needed in other applications, none are needed for this scenario. Data elements become available as a result of the introduction of devices that emit injury information, and are included in a rescue crew member's application as a result of context-maintenance performed to uphold the provided context definition. (It is important to note that while the application presented here requires only implicit insertion of data items by the infrastructure, the general context-sensitive data structures model is not limited to this type of insertion. A discussion of issues related to supporting insertion operations is presented in Section 4.)

In the disaster recovery application, the context-sensitive priority queue is populated with the victims in the context ordered by injury priority. The rescue crew member uses the application to get the head of the priority queue, dispatching a volunteer to tend to the victim until transport can be arranged. Because we consider that crew members may be assigned overlapping contexts and that the transport vehicles available to one crew member may not be available to another, the injury description obtained to dispatch treatment should still be made available. For this reason, the dispatch function of the application is implemented using the `getFirst()` operation previously described. Once treatment has been dispatched to the most severely wounded victim, the crew member uses the application on his PDA to determine if any transportation resources are available. If so, the application assigns the available transportation resources to the most severely injured victim in the context. Because the victim has been assigned on-site treatment and scheduled for evacuation, the victim should be removed from consideration by the rescue crew teams. Therefore, transport scheduling in the application should be implemented using the `removeFirst()` operation.

```
public class DisasterRecovery
   public DisasterRecovery()
      Context context =  one block radius
      PriorityQueue pq =
         new PriorityQueue(context);

   public void main(String args[])
      while(victimsUntreated())
         if(volunteersAvailable())
            TreatmentThread treat =
               new TreatmentThread(pq);
            treat.start();

         if(transportAvailable())
            TransportThread transport =
               new TransportThread(pq);
            transport.start();

   class TreatmentThread extends Thread
      the start method calls the run method...
      public void run()
         dispatch(getVolunteer(),
            (pq.getFirst()).id);

   class TransportThread extends Thread
      the start method calls the run method...
      public void run()
         assign(getTransport(),
            (pq.removeFirst()).id);
```

Figure 2. A CSDS Approach to the Disaster Recovery Application

Figure 2 shows sample code for an implementation of the disaster recovery application using a context-sensitive priority queue. This version of the application simply defines a context, instantiates the context-sensitive priority queue, and performs processing on the priority queue using the operations made available by the API, e.g., `getFirst()` and `removeFirst()`. The data structure does not have to be explicitly reconfigured by the application each time a victim is transported. Instead, an untreated victim in the context with the highest injury priority can be identified simply by using the `getFirst()` operation.

This example is suggestive of the programming productivity gains one could achieve with context-sensitive data structures. In the next section, we explore what is needed to support the implementation of context-sensitive data structures like the priority queue used in the disaster recovery application.

4. CSDS Infrastructure Support

We envision the gradual development of a library of context-sensitive data structures for use by context-aware application programmers. In most cases, the application programmer should not have to implement the context-sensitive data structure; she should simply choose among the available CSDS implementations. The application programmer is expected to use the API of the selected CSDS to interact with data as if it were local. Since many data structures share common operations, we envision providing an infrastructure that supports the development of context-sensitive lists, trees, stacks, queues, and other data structures.

At the heart of the CSDS model is the perception that we are populating a locally accessible structure with data items distributed throughout the reference agent's context, keeping the items in the local view consistent with the context as the environment changes. In reality, we are building a structure on top of the ad hoc network that mimics the organization imposed by a particular data structure. This overlay structure is used to support operations issued on a CSDS. As such, the structure must adapt accordingly in response to context changes.

To deliver a CSDS support infrastructure, we must explore what is required to build and maintain an overlay structure over the ad hoc network. To begin, we consider that the environment in which agents operate is open and dynamic. As the number of hosts that join the network grows, the number of context items available to an application agent significantly increases. Building an overlay structure to support the operation of an agent's CSDS over a large body of context items requires a substantial amount of processing. To aid in the development of efficient context-sensitive data structures, the infrastructure must contain protocols for limiting the scope of the context to include only those items that suit an agent's particular needs. The tailored context delivered as a result will be used by other protocols required in our infrastructure: those for supporting the implementation of particular operations on a CSDS. In the remainder of this section, we explore design issues associated with context scoping protocols and examine the effects of various CSDS operations on the development of protocols for building and maintaining overlay structures for ad hoc networks.

4.1 Protocols for Tailored Contexts

There are several viable approaches to limiting the reach of an agent's context. We utilize a policy-based approach similar to that in the network abstractions protocol [5] in which a context-scoping policy is used to determine an agent's context. In our approach, a context-scoping policy is associated with a particular CSDS. The policy is used to govern which context items in the ad hoc network are eligible for inclusion in the CSDS. An application programmer can specify the context associated with a CSDS by providing a context-scoping policy as part of the data structure's instantiation.

Each policy is defined as a set of constraints on properties of the ad hoc network. Constraints on properties of hosts (e.g., battery life), of communication links (e.g., bandwidth), of agents (e.g., access rights), and of data (e.g., type) may be used to define a context specification policy. We favor policy specifications that use constraints on such properties because they offer generality and flexibility and allow developers to reason at a higher level of abstraction about the entities within the ad hoc network and the way they contribute to defining the content of the CSDS.

The context-scoping protocol uses the scoping policy supplied by the application programmer to present a subset of the items in the ad hoc network as the content of the CSDS. In doing so, the protocol builds a context structure over the ad hoc network, which is then used by other protocols that support the execution of data structure operations. Certain scenarios call for different ways of using the context structure. When the environment is highly dynamic and data structure operations are issued over the context infrequently, the context structure is built on-demand each time that an operation on the associated data structure is issued. In situations where the environment is relatively stable and operations over the context are frequently performed, the context structure is maintained as the environment changes.

As a final note, implementations that supply agents with tailored contexts are implemented in a distributed fashion. Agents do not require global knowledge of the environment to participate in the computation of and to interact with their tailored context.

4.2 Data Structure Population Protocols

Typically, the insert operation described below is used to populate traditional data structures:

- `insert(X)`: places the data element X in the data structure according to its organizational policy.

Rather than allow programmers to insert data directly into a CSDS, the infrastructure performs data maintenance on behalf of the application. Thus, the `insert(X)` operation is not directly provided to programmers for a particular data structure in the context-sensitive data structures methodology. Instead, there are two ways to include data items in an application's CSDS: indirectly through context-specification and data element ordering, or directly by injecting data as a context item into the environment.

First, indirect insertion is used to populate a data structure. An application-provided context-scoping policy is supplied to the infrastructure in the instantiation of a CSDS. A protocol in the infrastructure for providing tailored contexts selects the data elements to be contained within the CSDS, and a separate protocol creates an overlay structure to mimic a local organization of those elements according to properties of the data structure. Like the context-scoping protocol, it may be practical in some situations to build the overlay structure on-demand when an operation is issued, or it may be more effective to maintain the overlay structure in the presence of changes. The overlay structure protocols often utilize several other protocols for data collection and aggregation. For instance, the implementation of the context-sensitive priority queue uses a protocol that sorts the items in the scoped context and returns the greatest element.

Second, direct insertion is performed by using an insertion operation provided on the *infrastructure* instead of on the context-sensitive data structures. We treat each data item produced by an application as a generic piece of data that is provided to the infrastructure via `insert` in order to supply it to other agents as context. The semantics of direct data insertion varies, and a suitable option can be specified by the programmer. We identify three types of direct insertion operations: local, destination-aware, and property-aware. In *local* insertions, the data item is stored locally by the inserting agent. A *destination-aware* insertion allows the programmer to specify a desired destination for a data item in terms of a particular agent or host. The inserted context item will eventually reside at the specified destination through the use of auto-migration. With auto-migration, the data item is delivered immediately when the destination is available. If the destination is unavailable (e.g., because of network partitioning), the data item is marked for migration and stored locally until it can be delivered. The *property-aware* insertion is similar, but allows a more decoupled method of specifying a recipient. With this type of insertion, a policy restricting the set of potential destinations is provided as a parameter to the insert operation. All destinations are evaluated against the criteria. A single destination is non-deterministically chosen from the set of matching destinations and is used by the infrastructure in a destination-aware insertion.

To allow agents to protect their data, specialized versions of the direct insert operations that support access control mechanisms can be used. Access control parameters are included with an insert operation to specify how the data is made available at different levels of protection. Only authorized agents are allowed to access or delete another agent's data items. The programmer can specify which agents are authorized using a policy similar to that used for context-scoping. This form of access control can be supported in part by requiring all agents to provide credentials in the context definition used to populate the data structure. These credentials are used by the context-scoping protocols to evaluate the access control rights against the access control policies of the provider to determine if the data is included in the context.

All of these insertion styles may affect the context associated with an agent's CSDS. If an overlay structure maintained for a particular context-sensitive data structure is affected by some agent's insertion operation, the protocol for maintaining the overlay structure must sense the change in the environment and accordingly adapt the structure.

4.3 Data Access Protocols

Data access in traditional dynamic set data structures can be generalized by the set of operations described below:

- `get(X)`: searches the data structure for the item corresponding to the key X. If successful, the operation returns the corresponding element; otherwise, it returns *null*.

- `contains(X)`: searches the data structure for the item corresponding to the key X. If successful, the operation returns *true*; otherwise, it returns *false*.

- `getNext()`: returns the next data element in the data structure. If the element does not exist, *null* is returned.
- `getFirst()`: returns the data element located in the first position of the data structure. If the element does not exist, *null* is returned.
- `iterate()`: returns an iterator over the data structure.

These operations do not change the data structure in any way. Thus, protocols designed to support data access operations simply use the overlay structure built by the population protocols discussed in the previous subsection. Examining the set of operations brings up questions about the semantics provided by the protocols. In some situations, an application's requirements may be satisified by weakly consistent results in exchange for more efficient execution of operations. In other scenarios, a strongly consistent reflection of the environment is required in the result, regardless of the expense of the distributed transactions needed for the operation's execution.

4.4 Data Removal Protocols

Manipulating the data structure by removing elements is a common task, and is typically achieved through the use of operations such as:

- `remove(X)`: returns and deletes the data element X from the data structure if it exists and adjusts the data structure if needed. If the element X does not exist in the data structure, the operation returns *null*.
- `removeNext()`: returns and deletes the next data element from the data structure if it exists and adjusts the data structure if needed. If the element does not exist, the operation returns *null*.
- `removeFirst()`: returns and deletes the data element located in the first position of the data structure if it exists, and adjusts the data structure; otherwise, the operation returns *null*.

Protocols developed to support these removal operations may have different semantics. We provide two types of removal operations: an *individual* remove and a *communal* remove. The former eliminates a data element from inclusion only for the issuing agent's particular CSDS on which the operation was called, while the latter expunges the data item from inclusion in any CSDS of any agent by removing the data item from the ad hoc network.

Individual remove operations are useful for collaborative applications that operate on overlapping contexts, such as the disaster recovery scenario presented earlier. In this application, once a rescue team member arranges treatment and transport, the victim is removed from the context-sensitive priority queue. However, volunteers and ambulance crews still require access to the injury information, and so it is not removed from the ad hoc network. To support individual removal operations, the protocol performs bookkeeping. When a remove is issued, the specified data item is marked as no longer belonging to a particular CSDS and that information is remembered by the owner as part of the data element. Choosing to use this protocol in the development of a CSDS, however, requires careful consideration, as the bookkeeping required can create a significant amount of overhead.

The more traditional communal remove operation eradicates the specified element both from the context-sensitive data structure and from the ad hoc network. This approach also requires careful consideration, since the protocol essentially deletes another agent's data. Using the access control approach previously mentioned, however, allows agents to control how other agents access their data.

Regardless of choice between individual or communal removal semantics, if the overlay structure is maintained, the removal operations require its restructuring. The removal of a piece of data is essentially a change in context and is handled by rebuilding the overlay structure when a data structure operation is issued over the context, or by the overlay structure maintenance protocol discussed in subsection 4.2.

4.5 Implementation Requirements and Issues

As we explore the potential for CSDS development, we make the observation that with particular data structures, the same portion is regularly accessed. For instance, priority queues and stacks are frequently accessed at the beginning positions of the data structures, using operations such as `getFirst` and `pop`. This observation is the motivation behind the concept of on-demand partially maintained data structures. We believe that improvement upon the performance of a typical CSDS can be facilitated by relaxing the requirement that the structure be built and maintained over all the data items in the context. Instead, a CSDS is initially constructed to consist of only the first n elements, where n is a parameter given in the instantiation of the data structure. As the elements are accessed, the structure is further constructed on-demand. The structure is maintained to n elements in the presence of context changes. The parameter n is tunable and can be changed to address the application's need or changes in the context. Our approach to partially maintained data structures has its roots in the *suspended cons* concept described in [4], which was introduced to support finite storage of infinite objects. Suspended `cons` is an extension to Lisp that allows placeholders of expressions to be stored until an operation forces its evaluation. Similarly, our partially maintained context-senstitive data structures are evaluated further only upon demand.

As a final note on supporting the development of context-aware data structures, it is imperative that we carefully evaluate the requirements and issues presented, develop protocols in response, and include them in an infrastructure. The delivered infrastructure should be flexible, allowing the CSDS programmer to use only needed components. The components should have minimal programming interfaces which are familiar and intuitive. The protocols should perform in a reasonably efficient manner. Moreover, the CSDS design methodology for context-aware application development should be put to the test through the development of applications that use context-sensitive data structures.

5. Conclusions

In this paper, we presented a novel abstraction called the context-sensitive data structure designed to simplify the development of context-aware applications. A context-sensitive data structure encapsulates data items distributed among a number of agents within a restricted portion of the large-scale ad hoc network and provides the programmer with access to the collection of data elements as if they were local through a well-defined API. The content of the context-sensitive data structure is fluid; as the context changes, the CSDS is reorganized to reflect the changes in the state of the environment. To support the use of context-sensitive data structures as a design methodology, we proposed providing an infrastructure that encapsulates protocols for restricting the context, for accessing data elements in the context, and for modifying data elements in the context. We envision this infrastructure as providing the CSDS developer with a set of essential tools that can be used to develop a range of context-sensitive data structures. In this paper we take a first step toward that goal by defining the context-sensitive data structures model and outlining the requirements and issues associated with developing a CSDS support infrastructure.

Acknowledgements

This research was supported in part by the Office of Naval Research under ONR MURI research contract N00014-02-1-0715. Any opinions, findings, and conclusions or recommendations expressed in this paper are those of the authors and do not necessarily reflect the views of the sponsoring agencies.

References

[1] G. Abowd, C. Atkeson, J. Hong, S. Long, R. Kooper, and M. Pinkerton. Cyberguide: A mobile context-aware tour guide. *ACM Wireless Networks*, 3:421–433, 1997.

[2] G. Cabri, L. Leonardi, and F. Zambonelli. MARS: A programmable coordination architecture for mobile agents. *IEEE Internet Computing*, 4(4):26–35, 2000.

[3] K. Cheverst, N. Davies, K. Mitchell, A. Friday, and C. Efstratiou. Experiences of developing and deploying a context-aware tourist guide: The GUIDE project. In *Proc. of MobiCom*, pages 20–31. ACM Press, 2000.

[4] R. Filman and D. Friedman. *Coordinated Computing – Tools and Techniques for Distributed Software*, chapter 12, pages 156–179. 1984.

[5] Q. Huang G.-C. Roman, C. Julien. Network abstractions for context-aware mobile computing. In *Proc. of 24th Int'l Conference on Software Engineering*, pages 363–373, 2002.

[6] A. Harter and A. Hopper. A distributed location system for the active office. *IEEE Networks*, 8(1):62–70, 1994.

[7] J. Hong and J. Landay. An infrastructure approach to context-aware computing. *Human Computer Interaction*, 16, 2001.

[8] C. Julien and G.-C. Roman. Egocentric context-aware programming in ad hoc mobile environments. In *Proc. of 10th Int'l Symposium on the Foundations of Software Engineering*, pages 21–30, Nov. 2002.

[9] A.L. Murphy, G.P. Picco, and G.-C. Roman. LIME: A middleware for physical and logical mobility. In *Proc. of the 21st Int'l Conf. on Distributed Systems*, pages 524–533, April 2001.

[10] N. Ryan, J. Pascoe, and D. Morse. FieldNote: A handheld information system for the field. In *1^{st} Int'l Workshop on TeloGeoProcessing*, pages 156–163, 1999.

[11] D. Salber, A. Dey, and G. Abowd. The Context Toolkit: Aiding the development of context-enabled applications. In *Proc. of CHI'99*, pages 434–441, 1999.

[12] R. Want et al. An overview of the PARCTab ubiquitous computing environment. *IEEE Personal Communications*, 2(6):28–33, 1995.