Programming Ad-hoc Networks of Mobile and Resource-Constrained Devices *

Yang Ni Ulrich Kremer Adrian Stere Liviu Iftode

Department of Computer Science
Rutgers University
Piscataway, NJ 08854
{yangni,uli,adrianst,iftode}@cs.rutgers.edu

Abstract

Ad-hoc networks of mobile devices such as smart phones and PDAs represent a new and exciting distributed system architecture. Building distributed applications on such an architecture poses new design challenges in programming models, languages, compilers, and runtime systems. This paper discusses SpatialViews, a high-level language designed for programming mobile devices connected through a wireless ad-hoc network. SpatialViews allows specification of virtual networks with nodes providing desired services and residing in interesting spaces. These nodes are discovered dynamically with user-specified time constraints and quality of result (QoR). The programming model supports "best-effort" semantics, i.e., different executions of the same program may result in "correct" answers of different quality. It is the responsibility of the compiler and runtime system to produce a high-quality answer for the particular network and resource conditions encountered during program execution. Four applications, which exercise different features of the SpatialViews language, are presented to demonstrate the expressiveness of the language and the efficiency of the compiler generated code. The applications are an application that collects and aggregates sensor data in network, an application that performs dynamic service installation, a mobile camera application that supports computation offloading for image understanding, and an augmented-reality (AR) Pacman game. The efficiency of the compiler generated code is verified through simulation and physical measurements. The reported results show that SpatialViews is an expressive and effective language for ad-hoc networks. In addition, compiler optimizations can significantly improve response times and energy consumption.

Categories and Subject Descriptors D.3.0 [*PROGRAMMING LANGUAGES*]: General

General Terms Design, Languages

* This work was partially supported by NSF-ITR/SI award ANI-0121416.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.

PLDI'05, June 12–15, 2005, Chicago, Illinois, USA.
Copyright © 2005 ACM 1-59593-080-9/05/0006…$5.00.

Keywords Ad-hoc Networks, MANET, Location-Awareness, Service Discovery, Quality of Result

1. Introduction

An ad-hoc network, a.k.a. MANET (Mobile Ad-hoc Network)[19], is a dynamic network spontaneously formed among mobile nodes without support from any infrastructure[38, 21, 8]. Ad-hoc networks are becoming a promising new target platform, with the proliferation of smart devices, i.e., small wireless devices with significant computing power, memory, and sensory capabilities. Typical examples of such devices are state-of-the-art smart phones and PDAs. These devices are able to provide information about their surrounding physical environment using sensors (e.g.: light, motion, temperature, pressure, speed), cameras, and microphones. Since not all their resources are used all the time, ad-hoc network nodes can potentially share cycles, memory, and sensors. The notion of sharing over a distributed platform has been successfully used in the context of peer-to-peer systems [13, 32, 24] and networks of workstations (NOW) [4]. Figure 1 illustrates an ad-hoc network. Smart phones and PDAs carried by pedestrians communicate via short-range wireless networking such as 802.11 or Bluetooth. Computer devices embedded in cars, buildings, or fixed structures such as a traffic light may also be part of the wireless network. The structure of the network changes dynamically as cars and people move, entering or leaving it. Additionally, the network shown in this figure may be part of a bigger ad-hoc network, perhaps spanning the entire city or some larger geographical area.

In contrast to the P2P and NOW distributed system architectures, the location of a network node in the physical space may be crucial for a distributed application executed on an ad-hoc network. Nodes are interesting for an application due to the hardware and software services that they provide and their particular location. Therefore, a programming model for ad-hoc networks must be able to describe spaces and desired services within such spaces. Since most such devices rely on battery power, any sharing will involve energy consumption, which will lead to a shortened battery life. As a result, applications should be able to set limits on their resource usage, trading off the quality of the produced answer for a reduction in the resource usage necessary to compute the answer. This tradeoff can be expressed as the desired quality of result (QoR). The quality of result is defined by the programmer and it is application specific in most cases. Due to the volatile and dynamic nature of the network, a program execution can return a range of "correct" answers, which must be partially orderable according to one or more quality criteria. An implementation of such a programming model should follow "best-effort" semantics to produce a high-quality answer under the particular network conditions,

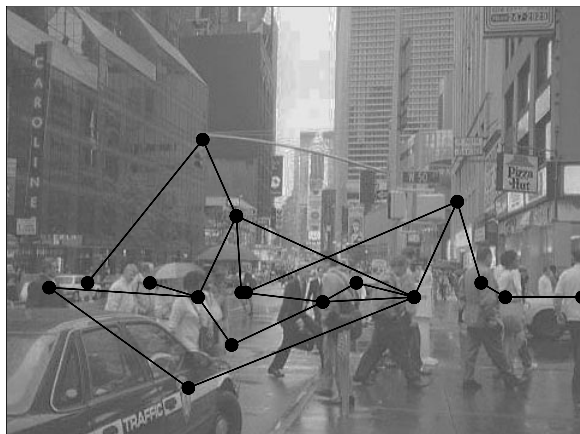


Figure 1. An envisioned ad-hoc network at a busy street corner.

while respecting user-specified resource constraints. Developing a formal “best-effort” semantics model is a new and important open problem, but is beyond the scope of this paper.

In this paper, we discuss SpatialViews, a programming language for ad-hoc networks. The goal of SpatialViews is to provide a high-level programming model, which allows application programmers to easily develop and maintain their ad-hoc network applications. Each node in the network is assumed to have substantial computation and communication capabilities, and is aware of its spatial location. The location of a node can be queried by the user level program, i.e., a particular location may determine the actions performed by a program. Severely resource-constrained sensor networks are not the main target of SpatialViews. Examples for target network nodes are state-of-the-art smart phones, PDAs, notebooks, and laptop computers.

The SpatialViews language is a vehicle to study different language, compiler optimizations, and performance/QoR/energy trade offs. Clearly, not all conceivable applications may be implemented within the framework of this programming model. Our goal is to provide a high-level programming model for a large class of applications for mobile ad-hoc networks that hides many details of the underlying volatile target networks. In this sense, SpatialViews is complementary to lower level languages such as nesC [14], SP [6] and SM [22]. A case study are presented in this paper, using four different applications implemented in SpatialViews. These applications are (a) an application that collects and aggregates sensor data in network, (b) an application that performs dynamic service installation, (c) a mobile camera application that supports computation offloading for face detection to a server discovered on-demand, and (d) an augmented-reality (AR) Pacman game that involves multiple players in a physical space “tagged” with virtual objects. The AR Pacman game application was inspired by the Human Pacman project[11] and the Pac-Manhattan game[37]. These applications exercise different SpatialViews features and have been chosen to illustrate the expressiveness of the language, and the effectiveness of our prototype compiler to produce efficient code. The performance of the compiler generated code were evaluated through simulation and physical measurements. Other applications such as TrafficView[12] and EasyCab[49] can also be implemented in SpatialViews, but we did not evaluate them for this study.

In our previous work[35], we provided a straightforward serial implementation of SpatialViews. A single program “walks” around the network and executes on each encountered node, one node after another. The serial implementation has scalability and robustness issues. When the network size increases, the execution time increases linearly at best. Even partial network disconnection or failures of single network nodes may lead to the loss of the program, resulting in no answers being reported. In [36], we performed an initial benefits study of different parallelization and replication techniques and their impact on program response time, energy consumption, and quality of result. The discussed techniques were based on flooding, spanning-tree construction, and bounded program replication. Only a single sensor network application was used in that previous study.

The contributions of this paper are as follows:

1. A detailed discussion of the SpatialViews language and its features.
2. An evaluation of the expressiveness and performance of SpatialViews and its prototype compiler for four application programs that stress different features of the language. Evaluation is done in terms of QoR, response time, and energy consumption. Physical measurements were performed on a network of 12 HP iPAQ handheld PCs running Linux and communicating through a wireless 802.11 network connection. Simulation results were obtained for up to 64 network nodes.
3. A discussion and evaluation of compiler optimizations used to implement a core SpatialViews construct, namely the spatial view iterator.

The reported results show that the language model is expressive and allows efficient implementations. The authors are very aware that the success of any new language model will ultimately be measured in terms of its acceptance by users who try to run applications they care about on ad-hoc networks. A first version of the SpatialViews compiler, runtime system, and debugging/visualization environment is publicly available[1]. We believe that this paper makes a strong case for viewing ad-hoc networks as an interesting distributed computing target platform with exciting and wide-open application potential. Part of this potential can be easily accessed through the use of our SpatialViews programming language. In addition, SpatialViews has proven itself as a research infrastructure into new compiler optimizations for ad-hoc networks.

2. SpatialViews Language

The programming target of SpatialViews is a network of nodes embedded in the physical world. A node in the physical world may be of interest because it is at a specific place (location) at a specific time, providing a specific service. Therefore, location and time are crucial concepts for a programming model for ad-hoc networks, in addition to node functionality. A first-class abstraction in SpatialViews is a virtual network explicitly named by the services and locations of its nodes, and instantiated across time. The actual network embedded in the physical world consists of many such virtual networks. For each virtual network, computation is specified. The computation is performed on individual nodes of the specified virtual network, with computation migrating from one node to another. It is desirable to support computation replication and parallelization, which allows computation to be performed on several nodes or migrate between nodes at the same time.

A virtual network is declared as a *spatial view*, and instantiated using a *spatial view iterator*. A spatial view declaration requires a set of services and a space. An iterator instantiates a spatial view by discovering those nodes that provide the services and reside in the space, and by migrating computation to them. The iteration procedure may be limited by a time constraint which represents a time budget that once expired will lead to the termination of the iteration procedure. The virtual network is thus the collection of nodes that provide the specific services and are confined to a

[1] SpatialViews web site http://www.cs.rutgers.edu/spatialviews

space-time region defined by a spatial view and an iterator. To write a program in SpatialViews is to define spatial views and their iterators. SpatialViews is an extension to Java. Computation is defined in each iterator as in standard Java.

For the remainder of this paper, a node is assumed to be a virtual node unless explicitly specified otherwise. A virtual node is bound to a physical node within a space×time region. SpatialViews allows the specification of space and time granularities of this space×time region. These granularities are closeness metrics in terms of space and/or time, respectively. Every node in the virtual network has to be distinct from any other node either in terms of space or time. This language feature allows the programmer to control the "density" of a virtual network that is spread across time and space, making tradeoffs possible between performance, energy consumption, and quality of result (QoR).

As time changes, the same physical node may represent different nodes in a virtual network. If a physical node occupies multiple locations due to its mobility, it may also represent different nodes at the same time, i.e., within the same time granularity. The finest granularity that can be chosen by an application in terms of time and space depends on the location and timing technologies available in the physical network.

No bindings between virtual nodes and physical nodes can be made explicit or permanent, because such bindings may change over time and space. For this reason, a virtual network can only be accessed by dynamically instantiating a spatial view using an iterator. A more detailed description of the SpatialViews language can be found elsewhere [26].

2.1 Spatial View Definition

A *spatial view* is a collection of virtual nodes each of which provides a given set of services and resides in a given space. A spatial view defines a virtual network over the real, physical network. A *virtual node* is the programming abstraction for a physical node, which can be denoted as a tuple (services, location, time). The services are provided by the physical node. The location is the location of the physical node. The time is the time when the program starts its execution on the physical node representing the virtual node. A virtual node is an execution environment in which a program has access to the denoted services at the denoted location and time. To simplify our discussion, we will assume for the remainder of the paper that each virtual node specifies only a single service.

A service is named by a Java interface. The name of the interface, the method list of the interface, and the semantics of all the methods are agreed upon by all the participants of the network. An object is said to provide a service if it implements its interface. A physical node is said to provide a service if it hosts an object that does. A physical node may provide multiple services, either by hosting multiple objects or one object implementing multiple interfaces. Such a physical node is represented by more than one virtual node. Installation of services will be discussed in Section 2.3.

A space is represented by a space type object. A space type is class `Space` or its derived class. Class `Space` has a single abstract method `contains`, which takes a location as an argument and returns a boolean value – true if the space contains the location, and false if not. The derived classes of `Space` include `Circle`, `Rectangle`, etc. `SpaceUnion`, `SpaceIntersection`, and `SpaceDiff` – derived classes of `Space` – allow composition of complicated spaces using simple spaces. A location is represented as a `Location` object. Derived classes of `Location` may represent locations in different formats, including GPS locations, MIT Cricket locations[39], or "fake" locations appropriate for a particular simulation purpose. Usually, such a `Location` object is returned by querying the location service. The current implementation of the SpatialViews runtime library supports GPS and Cricket locations. For instance, a call to method `createGPSLocation` creates a `Location` object from longitude, latitude, and, optionally, altitude.

A space granularity Δs can be defined for a spatial view. It defines the spatial density of the virtual network. Specifically, the expected density for a spatial view with space granularity Δs is $O(\Delta s^{-2})$ for 2D spaces, i.e., one node every Δs^2 area. In 3D spaces, the expected density is $O(\Delta s^{-3})$. Beyond density, the specification of Δs requires virtual nodes to be approximately uniformly distributed across the entire space. Space granularity gives the programmer control over QoR, and provides the compiler and runtime system an opportunity to make tradeoffs among performance, resource usage, and QoR.

Spatial views are defined using the `spatialview` statement. The following statements define two spatial views. Space granularity Δs is defined using a `%` (per) operator in a `spatialview` statement.

```
spatialview sv1 = Camera @ BuildingC.Floor3;
spatialview sv2 = LightSensor @ CampusB % 100;
```

`Camera` and `LightSensor` are class names for two Java interfaces. `BuildingC.Floor3` and `CampusB` are variable names for two space type objects. They are defined as following.

```
Rectangle CampusB=new Rectangle(...);
class CBuildingC extends Rectangle {
    //constructors omitted
    ...
    //arguments omitted
    public Rectangle Floor3=new Rectangle(...);
}
CBuildingC BuildingC = new CBuildingC(...);
```

These space definitions can be put into a library and imported into an application program. For spatial view `sv2`, the space granularity is set to be 100 meters. For `sv1`, since no space granularity is explicitly specified, it is assumed to be the finest accuracy needed to distinguish two physical nodes, or the finest achievable with the available positioning technology.

2.2 Spatial View Iterator

Once a spatial view is defined, an iterator can be applied to it. The iterator discovers the virtual nodes in the spatial view, gets access to their services, and migrates program execution to them.

In SpatialViews, an iterator is expressed as a `visiteach` statement. For example,

```
visiteach x : sv1 {
  Picture p=x.getPicture();
  ...
}
```

where `sv1` is the spatial view of cameras defined above, and `x` is an object which is an instance of the service (defined by the `Camera` interface in this example). When the control flow reaches the `visiteach` statement, the program should have migrated to a node that provides the service and is in the specified space(`BuildingC.Floor3` in this example), and `x` should have been initialized as an object of type `Camera` to access the service. The program continues its execution on the new node, until the end of the iterator. One execution of the iterator body on a node is called a visit to the node. After visiting one node, the program will try to visit another virtual node. If no more nodes can be found, the program will migrate back to the node where it migrated from before the program execution reached the iterator. The order in which the program visits the nodes is not known a priori. In fact, the program may replicate itself and execute on multiple nodes in

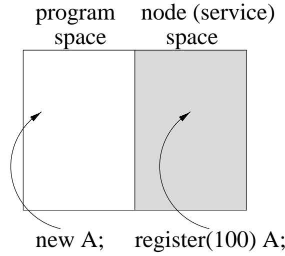


Figure 2. The SpatialViews memory model: The program space is part of a program's execution state and migrates with the program; the node or service space is a resilient memory area and does not migrate. Objects in program space are allocated using the `new` construct. The `register` creates objects in the service space. Service variables have a specified, maximal life time (100 seconds in this example) and can be accessed by other SpatialViews programs, thereby allowing exchange of information between programs.

parallel. However, only one instance of the program remains after the iterator has finished.

During an iteration, unique virtual nodes are visited. Uniqueness requires that the location or time of a virtual node differ from other nodes. The difference should be at least an amount specified by the space or time granularity. Space granularity is specified in the spatial view definition. Time granularity is specified in the iterator. Different physical nodes can be visited, as long as they have distinguishable locations, or are visited at different times. A single physical node can be revisited in Δt time after the previous visit, where Δt is the time granularity. It can also be revisited if it changes its location at least Δs, where Δs is the space granularity. In practice, the desired spatial and temporal distribution may be approximated. The current implementation uses a relaxed model that considers virtual nodes different if they are in different $\Delta s \times \Delta t$ subgrids of the original space$\times$time target region.

Infinite iteration is possible, because the target virtual network could be infinite. There are two reasons. First, new physical nodes may join the network. Second, with time change or movement, the same physical nodes can be reused, and keep appearing as new virtual nodes in our spatial view. Although infinite spatial view iterations have their applications, it is often desirable to avoid them. The time constraint on an iterator can be used for this purpose. A time constraint is a time budget for an iteration. After each visit to a virtual node, the remaining budget will be checked. If the budget is exhausted, further visit to new nodes will be prevented.

The time granularity Δt is specified in the `every` clause. An `every` clause is always followed by a `within` clause or `forever` keyword. For example.

```
visiteach x:sv1 every 3 within 600 {...}
visiteach x:sv1 every 5 forever {...}
```

where "every 3 within 600" means: $\Delta t = 3$ seconds and the time constraint is 600 seconds, and "every 5 forever" means: $\Delta t = 5$ seconds and the time constraint is infinity. A `within` clause or a `forever` keyword is always used after an `every` clause and never used alone. If no `every`, `within`, or `forever` clauses are used in the iterator, the programmer indicates that as many virtual nodes as possible should be visited without using a physical node more than once. In other words, an unqualified iterator produces a snapshot of the network that can be greedily discovered in one try.

2.3 Memory Model

Every variable in a SpatialViews program is either a *program* variable or a *service* variable. Program variables migrate as part of the program execution state from one virtual node to another. Service variables do not migrate. The main motivation for service variables is the support of cooperation among SpatialViews programs, and the access to services provided by and residing on physical nodes, e.g. hardware-specific functionality such as access to sensors or cameras. Program variables are stored in the program space, and service variables are stored in the node (service) space. Program variables are created using the `new` operator or are of basic type, and service variables are allocated using the `register` operator. Figure 2 shows the partitioned memory model for SpatialViews programs.

Service Variables A service variable is declared in an iterator. It can only be accessed in that iterator, but not in any nested iterators. A `register` operation creates a service object in the node space from a class that implements one or more service interfaces. That service object can be used later through a service variable. For any created object, a `register` operation generates one or more entries in the service table of the hosting node. The service table is a data structure that maps service names (interfaces) to service objects. An object's names are all the interfaces that it implements, directly or through inheritance. Once created, a service object can be found by an iterator through its names in the service table, and can be bound to a service variable. The `register` operator contains a parameter that specifies the lifetime of the created object in seconds. This lifetime is a hint for garbage collection in the node space. The runtime system does not guarantee that the registered service will be available for its specified, maximal lifetime. The host node may decide at any time to temporarily suspend access to service variables, or to permanently delete service variables.

Program Variables There are three categories of program variables in a SpatialViews program. Each category has specific access constraints, giving opportunities for different optimizations such as parallel execution of the iterations of a spatial views iterator using structured communication patterns [36]. Without such restrictions, race conditions could occur during parallel execution. By default, all program variables are assumed to be local. SpatialViews program variables can be of one of the following categories:

1. local: a local variable is read/write within the defining iteration, and read-only within nested iterators.
2. container: a container variable represents a collection of objects. It is read/write within the defining iteration, and write-only within nested iterators. The corresponding abstract data type is that of a set of elements of a particular type. "Write-only" means that objects can only be inserted into the collection, but not read or removed. A container variable must be an instance of the predefined `Container` class.
3. reduction: a reduction variable is specified together with a commutative and associative operation. It is read/write within the defining iteration, and apply-reduction-operation-only within nested iterations. The initial SpatialViews language will only support a rather small subset of reductions, such as sum and product reductions. Reduction variable declarations start with the keywords `sumreduction` or `productreduction`.

There are no global, shared variables without any access restrictions in the SpatialViews language. If a variable cannot be classified as either one of the three types of program variables or as a service variable, a compile-time error will occur. The compiler and runtime system will enforce the access restrictions for the program variables.

```
public interface LightSensor {
  public float read();
  ...
}
public interface SpaceDefs {
  public static final Space CampusB=new Rectangle(...);
  ...
}
public class AverageLighting {
  public static void main(String[] args) {
    sumreduction float s=0;
    sumreduction int n=0;
    spatialview sv=LightSensor @ SpaceDefs.CampusB % 320;
    visiteach x : sv
      { s += x.read(); n++; }
    if (n>0)
      System.out.println(Float.toString(s/n));
  }
}
```

Figure 3. An average sensor reading program.

2.4 Example Program

Figure 3 shows an example program that collects readings from light sensors and calculates the average. The program contains a single spatial view that specifies a virtual network of light sensors on a university campus, with a desired density of one sensor every 320×320 square meters. The iterator declares a service variable `x`. When the program visits a node, `x` will be bound to an object that implements the `LightSensor` interface on the visited node. The service object lives in the node space, allowing accesses to the light sensor installed on the node. In this example, the space definition is assumed to be provided by the space definition class `SpaceDefs`. This class may be written by the user or may be imported from a library of space definitions. Since the iterator does not contain any clauses, the program will try to find and migrate to as many virtual nodes as it can while observing the space constraint, and then return to the program injecting node, i.e., the machine that started the execution of the entire program. The body of the iteration, namely {`s += x.read(); n++;`}, is executed on each visited virtual node. Both `s` and `n` are reduction variables. They are allocated in the program space and migrate with the program. Since the values of the reduction variables cannot be read in the iterator, the compiler has the option of generating code that exploits the parallelism in this program, for instance, using of geographic flooding as discussed in Section 3.1.2. Once the iterator terminates, the values of `s` and `n` will have been appropriately updated. The average light sensor reading is calculated from `s` and `n`, and printed on the injecting node.

3. Implementation

The current implementation includes the SpatialViews compiler, virtual machine, runtime library, and debugging/visualization environment. The runtime library implements different iteration approaches that are selected through compiler options. Based on the selected iteration approach, the compiler generates code that extends classes in the appropriate library. The current prototype compiler performs type checking for access restrictions on program and service variables, but does not support any interprocedural analysis yet. The virtual machine provides support for program migration. Figures 4 and 5 show the compiler and runtime system, respectively. The debugging and visualization environment is a program that emulates ad-hoc networks of mobile nodes, on which a programmer can run and debug compiled SpatialViews programs before deploying them to real networks.

3.1 Compiler

The prototype SpatialViews compiler extends `javac` in Java Development Kit (JDK) 1.3.1 from SUN Microsystems, Inc. A flow graph is given in Figure 4 for the compiler. The parser was changed to accept new statements such as `spatialview` and `visiteach`, and the corresponding new intermediate representations (IR) were added. Program analysis or transformation at the IR level are implemented as translators (a.k.a. passes).

The SpatialViews compiler added two main translators to the SUN Java compiler. One translator verifies that variables are accessed as declared in an enclosed iterator, i.e., non-local variables are never written, reduction or container variables are never read, and service variables are never written or read. If this check fails, a compile-time error will be reported. The other translator performs optimizations such as parallelization of spatial view iterators based on the user selected iteration strategy. For all updates on reduction variables and container variables, the translator generates code that does local computation and code that merges partial results. In addition, this translator generates code to implement transparent program migration on top of the SmartMessages[22] virtual machine.

3.1.1 Migration

SpatialViews supports transparent program migration. The current implementation does not allow recursive calls from within an iterator. Migration is implemented at two levels. At the lower level, the SmartMessages[22] virtual machine supports explicit migration. At the upper level, the compiler generates code to make migration appear transparent.

The SmartMessages system is an extension to the Java 2 Platform, Micro Edition (J2ME)[41]. The Kilobyte Virtual Machine (KVM)[42] and the Connected, Limited Device Configuration (CLDC) class library were modified to implement light-weight migration. J2ME has been widely used in today's cell phones. The memory budget is in the range of 160KB to 512KB.

The migration provided by SmartMessages is explicit, opposite to transparent full-process migration. Only a very limited amount of information about the program execution state, including the instruction pointer, stack pointer, etc., is automatically transferred to the new node. No program data are automatically transferred. Instead, a collection of data are explicitly allocated (called a data brick) for the program to carry in a migration. If the value of some variable is needed after migration, the program should copy it into the data brick. It is the SmartMessages program's responsibility to restore the variable with the value from the data brick after a migration. At the upper level, the SpatialViews compiler transparently decides what data items have to be migrated, packs and unpacks the resulting data bricks, and thereby hides all the above discussed details. As a result, at the SpatialViews language level, a programmer does not explicitly deal with program migration, and no migration primitive is provided.

3.1.2 Iteration

Spatial view iteration is end-to-end migration among the injecting node and virtual nodes that provide the service in the space. Different iteration approaches can be chosen with a command line option for the SpatialViews compiler. Each approach represents a different tradeoff among performance, resource usage, and QoR. The set of currently supported approaches is discussed below.

Simple Iteration approaches do not make use of location information in the process of searching for new nodes and the routes to them. Location information is only used to decide whether a found node is part of the spatial view or not. A straightforward implementation of spatial view iteration is a serial implementation as discussed in [35, 36]. In a serial iteration, a single program migrates from one node to another. To improve its failure resilience or

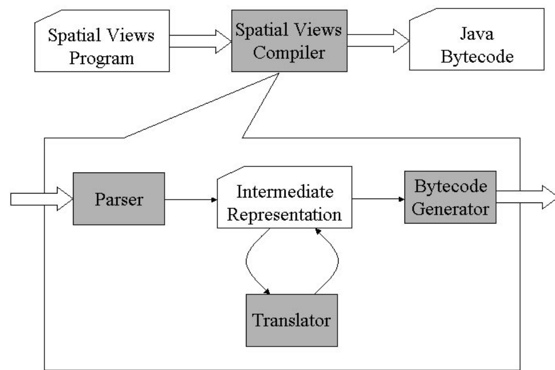


Figure 4. The SpatialViews compiler.

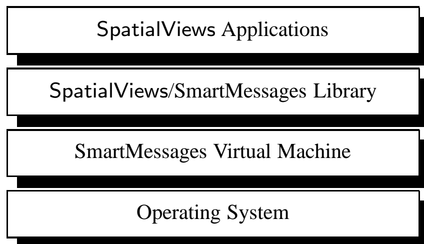


Figure 5. The SpatialViews runtime environment.

performance, a program may be replicated. The replica may work independently, each of which performs a serial iteration, or they may cooperate with each other. Experiments showed that cooperative approaches are more failure resilient and more energy efficient. In a cooperative approach, a program clones itself on a new node. The clones are propagated to nodes immediately (one hop) reachable from the current node. The clones then clone themselves and propagate to more nodes, and so on. The clones mark nodes so that they do not visit a node that has been visited by another clone. When a limit in the node discovery strategy is reached or the time constraint exceeded, the clones migrate back. The partial results of reduction variables and container variables are merged on intermediate nodes while clones are converging back. After the partial results are merged, the clones that produced the partial results terminate. In the end, the original program continues on the node that initiated the iteration and all clones terminated. The program has the final results of all reduction and container variables. This approach is called *flooding*. A *tree-based* approach is similar to flooding, but remembers the spanning tree of visited nodes for the spatial view. Subsequent iteration over the same spatial view will reuse the spanning tree. The tree-based approach can be more efficient than a flooding approach. However, the tree-based approach does not allow the discovery of new nodes and/or routes. A more detailed discussion of these approaches can be found elsewhere [36].

Geographic Iteration approaches use location information. Two geographic iteration approaches are proposed in this paper. Both approaches use a quadtree[40] to represent the target space. A *quadtree* is a recursive division of a minimal square cover of the target space. For 3D spaces, *octree*s can be used. However, a discussion of octrees is beyong the scope of this paper. To construct a quadtree for a space, a minimal square cover has to be found. This square is divided into four smaller squares. The four smaller squares are divided into even smaller squares, and so on. This division process continues until the size of the smallest squares is less than Δs as defined in the spatial view. The smallest squares in this division are called *cells*. The cells that are not located the original target space can be ignored. The largest square is represented by the root of the quadtree. It has four children, which are four smaller squares that the root is divided into. They are at the first level of the quadtree. The smallest squares (cells) are the leaves of the quadtree. Given the quadtree, a geographic iteration should visit one node in each cell, if there exists such a node. This way, the density of the virtual nodes is $O(\Delta s^{-2})$, and they are evenly distributed over the target space.

In the *Geographic Serial Iteration* approach, an iteration is modeled as a multi-stage dynamic planning problem and solved using a greedy strategy. The program remembers all visited cells. After visiting every node, the program migrates to a neighbor, if and only if that neighbor minimizes the program's distance to unvisited cells. That distance is defined as the shortest among the distances from the program's running node to all unvisited cells. As a special case, if a physical node is located in an unvisited cell, this distance is zero. After the program has visited all the cells in a quadtree, it migrates back to the injecting node, using geographic routing. Backtracking maybe necessary in sparse networks. If a cell does not contain any virtual node, the algorithm will only detect this fact after exhausting all possible alternative routes to reach a node in the cell. This results in a significant overhead and may make this approach less efficient than simple serial iteration.

Geographic Flooding, the second approach, is similar to simple flooding. The difference is that the program is propagated over spatial topology instead of network topology. Specifically, the program first migrates into the root of the quadtree, i.e., any node in the minimal square cover of the target space. Subsequently, the program forks into four clones, each of which migrates to the next level of the quadtree, i.e., the sub-squares of the current square. This process repeats recursively until the program clones propagate into all the cells. If the first physical node visited by the program in a cell does not provide the service, a simple serial iteration confined to that cell has to be performed to find another physical node that does. And this extra serial iteration will stop on the first of such nodes. Finally, all the clones migrate up the quadtree back to the starting node. Partial results are merged on the way back. In this approach, geographic routing is used to migrate from one node to a specified square region.

3.2 Programming Environment

Our SpatialViews development environment also contains a debugging/simulation/visualization component designed to facilitate testing SpatialViews code on an emulated mobile ad-hoc network. Mobility is emulated by feeding dynamically generated location and topology information to a collection of KVM processes running on the same PC. The debugger can be used to inject code into the network, observe its behavior, and interact with the system through a graphical user interface. The same compiler generated code can be executed on a real target system, for instance a collection of HP iPAQs, or on an emulated network using the debugging/simulation/visualization environment.

Currently, the debugger can display a schematic view of the network topology and tracking the movement of individual nodes, inject SpatialViews code into the network through a user-selected node, and display program output and migration information. Planned features include the ability to recreate real-life network configurations based on location data gathered experimentally, more extensive control over the parameters of the synthetically-generated network configurations, and a more structured presentation of debugging information gathered from KVM processes.

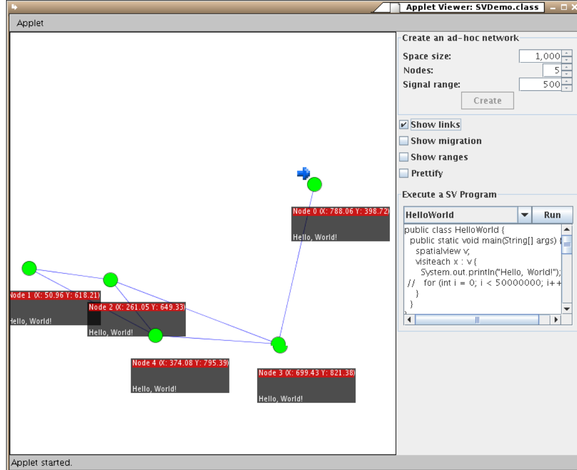


Figure 6. Debugger/Visualization Prototype for SpatialViews

A general "replay" mechanism will allow the understanding and debugging of a particular program execution. Such a replay mechanism is important since the particular program/system behavior may not be reproducable.

A preliminary version of the debugger which allows remote real-time interaction with several example SpatialViews programs running on an emulated network hosted on our servers is available as a Java applet. Figure 6 shows a screen snapshot of the current implementation. Each gray dot represents a node in the emulated network. Each node has an associated display area which shows the node's spatial position (top) and the last few lines of that node's output. The user interface panel on the right allows the user to (1) control several parameters of the network, (2) select a program from a list of compiled SpatialViews programs available on the system and inject it into the network, and (3) overlay information about signal ranges and code migration paths onto the main display window.

4. Experimental Results

We will illustrate the expressiveness of the SpatialViews programming model and the performance of the compiler generated code based on four example applications. Each application exercises different features of the language.

4.1 QoR vs. Resource Usage Tradeoffs

Most SpatialViews target applications are location sensitive, i.e., location information is required by the applications. In some cases, it may be unnecessary or wasteful to visit every node in a dense network if an acceptable program answer can be computed by only visiting a representative subset of the network nodes. For example, this representative subset are nodes that are evenly distributed across the target space. As a rule of thumb, the fewer nodes are visited, the faster the program will return and the lesser network and node resources are used. This comes at the price of a potential reduction in the quality of the produced answer.

The space granularity in a spatial view definition allows the user to express a QoR vs. resource usage vs. performance tradeoff. We evaluated both geographic approaches with simulations and experiments. The reported results used the light sensor program as shown in Figure 3. The experimental platform was a network of 12 H3700 or H3800 iPAQs each running the SmartMessages virtual machine under Familiar Linux. The simulation environment was a PC running 64 KVMs under RedHat 9 Linux. The KVMs in the simulation environment were the same as those running on the iPAQs except that they were built for the x86 ISA. The compiler generated the same bytecode for both simulations and experiments.

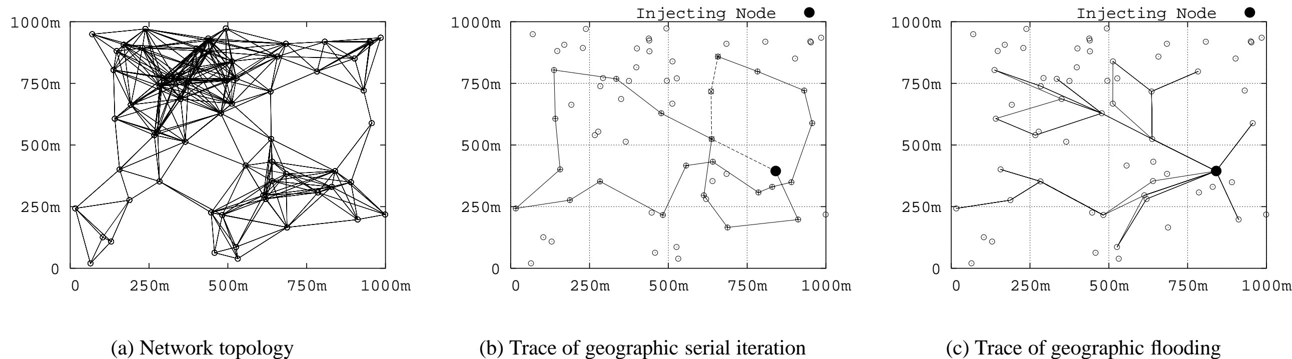
The simulated network topology is shown in Figure 7(a). The target space was 1000m$\times$1000m. The nodes were randomly distributed over the space with a wireless network signal range of 250m. Each pair of nodes that are within the signal range of each other are connected with an edge in the figure. Figure 7(b) shows the trace of the program migrating with the geographic serial approach. Figure 7(c) shows the trace using geographic flooding. Isolated nodes without any edges were not visited at all. Only 24 nodes were visited using either approach. If the simple serial or network flooding approaches are used, all the 64 nodes will be visited. These simulation results imply that geographic approaches will improve response time and save energy consumption over simple iteration approaches.

We did physical measurements of energy consumption and response time on 12 HP iPAQs using both geographic and simple iteration approaches. We connected the 12 iPAQs to a single DC power supply in parallel. An oscilloscope was used to measure the current at the output of the power supply. The current readings were collected by a data acquisition PC. Figure 9 shows a diagram of the testbed setup. We calculated the power dissipation of all the iPAQs from the current readings and the input voltage of 5V. Since all batteries were fully charged before the experiments and since the iPAQs' DC/DC converters are highly efficient, the observed power dissipation is very close to the actual power dissipation of the network. Energy consumption was calculated by integrating the power over the execution time. In the reported results, energy consumption of idle state was deducted. In other words, only the extra dynamic energy consumption caused by the program execution is reported.

Since all iPAQs were connected to a common power supply in the experiments, their physical distribution was limited by the length of their power cables. As a result, all iPAQs were able to directly communicate through their wireless connection. Even when we shielded each individual iPAQ with metal foil, the radio signals were still able to travel along the power supply cable to all iPAQs. To get a more interesting network topology, we disabled the dynamic network neighbor discovery in these experiments, and instead used static configured neighbor lists. Figure 8(a) shows the used network topology. The target space was 625m$\times$625m with a wireless network signal range of 250m. The node locations were simulated, i.e., statically configured. The program was injected from a laptop computer which is not shown in the figures. Figure 8(b) shows the trace for geographic serial iteration, and Figure 8(c) the trace for geographic flooding. Figure 10 reports energy consumption, response time, and number of visited nodes for the four approaches. Compared to simple iteration, geographic iteration saved 50% or more energy and ran at least twice as fast in our experiments. These savings were achieved by visiting only a spatially representative subset of nodes that cover the entire target space. Depending on the application, this may only slightly impact the quality of the produced result. Our example application has this property. As pointed out in Section 3.1.2, the effectiveness of geographic iteration also depends on the network density and may not work well for sparse networks.

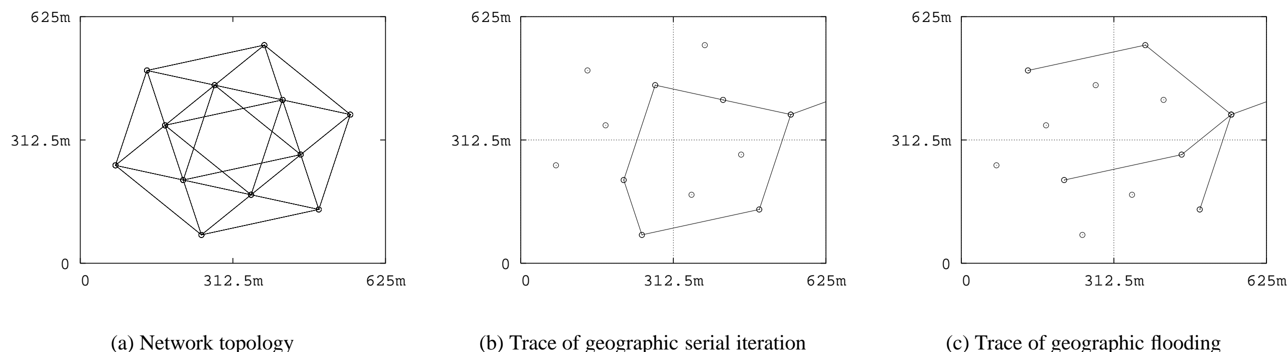
4.2 User-Defined Services

An `every` clause in a spatial view iterator allows a physical node to be visited again as a new virtual node every Δt time interval. This feature can be used to specify and deploy a user-defined service which provides automatically refreshed information within every Δt time interval. Figure 11 shows such a SpatialViews program that installs and updates a location service. The program installs an



(a) Network topology (b) Trace of geographic serial iteration (c) Trace of geographic flooding

Figure 7. Simulation on 64 nodes with the average sensor reading program (Figure 3).



(a) Network topology (b) Trace of geographic serial iteration (c) Trace of geographic flooding

Figure 8. Experiments on 12 iPAQs with the average sensor reading program (Figure 3).

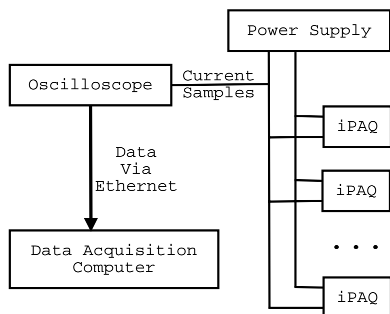


Figure 9. The energy measurement setup.

"eager" location service on every node within a hallway that periodically queries a positioning system based on MIT Crickets [39]. The Cricket system may take up to four seconds to acquire a location reading. If a program running on a node in the hallway needs to know its location, an on-demand, i.e., lazy query of the Crickets may lead to a significant performance bottleneck, for instance during the execution of a spatial view iterator [36]. The latency of the Crickets may be hidden from an application by using an eager location service instead. The eager service can be written and deployed by the SpatialViews programmer, i.e., the user.

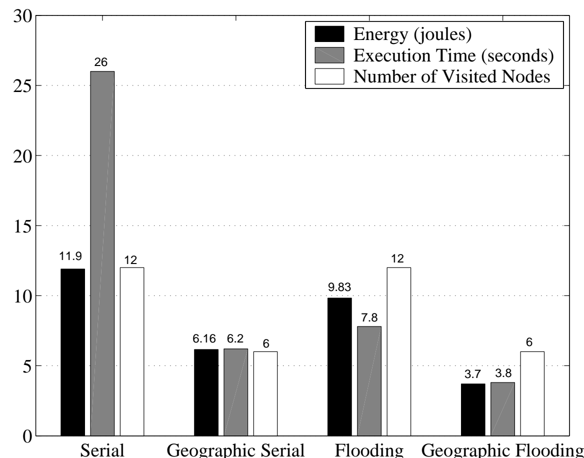


Figure 10. The measurements for the average sensor reading program running on 12 iPAQs.

```
public interface LocationService {
  public Location currentLocation();
}
public class EagerLocationService
       implements LocationService {
  Location l;
  public EagerLocationService(l) {this.l=l;}
  public Location currentLocation()
    { returns l; }
}
public class DedicatedLocationService {
  public static void main(String[] args) {
    float dt = Float.valueOf(args[0]).floatValue();
    spatialview sv = @ Hallway;
    visiteach x : sv every dt forever {
      CricketLocationService ls=
        new CricketLocationService();
      Location loc=ls.currentLocation();
      register(dt) EagerLocationService(loc);
    }
  }
}
```

Figure 11. A user-defined service example.

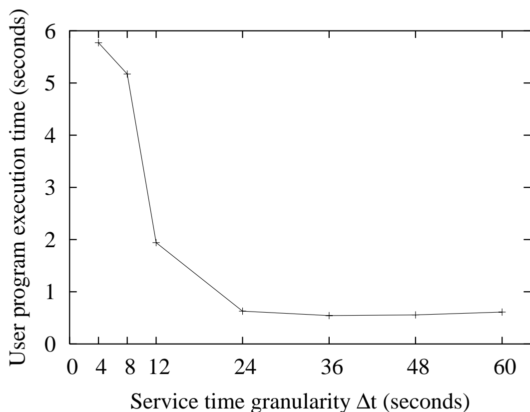


Figure 12. Overall execution times of the average sensor reading program (Figure 3) on 6 iPAQs using the user-defined eager location service (Figure 11) with different time granularities Δt (= `dt` in Figure 11).

We conducted experiments with this eager, self-updating location service. We ran the service on 6 iPAQs. As an application program, we ran the light sensor program shown in Figure 3 using flooding and a fully-connected network topology. We measured the overall execution times of the light sensor application using the eager location service with Δt values ranging from 4 seconds to 1 minute. The results are reported in Figure 12. When the application used the Cricket service directly, i.e., in a lazy fashion, the execution time was 8 seconds. In contrast, the execution times using the eager service were significantly lower, ranging from 0.6 to 5.8 seconds. The eager service will compete with the application for CPU time. The more frequently a service runs, the less responsive the overall system will be and the more resources will be used. From a programmer's perspective, the choice of Δt represents a trade-off between QoR ("freshness" in this case), system responsiveness, and resource usage.

```
public class ShootAndDetect {
    public static void main(String[] args)
    {
        Container result = new Container();
        spatialview CameraView = Camera @ BuildingC.Floor3;
        visiteach c : CameraView {
          Picture p = c.getPicture();
          spatialview DetectorView = FaceDetector;
          visiteach d : DetectorView {
            result.addElement(d.detect(p));
          }
        }
        int i=0;
        for (Enumeration e=result.elements();
             e.hasMoreElements();) {
          Picture p=(Picture)e.nextElement();
          p.savePNMFile("PIC"+(i++)+".pnm");
        }
    }
}
```

Figure 13. A camera and face detection example.

4.3 Cooperating Nested Virtual Networks

Multiple spatial views cooperating with each other is a very useful feature of SpatialViews. This feature is usually expressed via nested iterators. Based on conditions encountered during program execution, additional service discovery is initiated. Figure 13 shows a SpatialViews program that finds nodes with cameras within a building, instructs the cameras to take pictures, and then initiates face detection on server nodes that provide face detection.

We ran the program on the 12 iPAQs shown in Figure 8(a) and a laptop computer. Only one iPAQ had a camera sleeve and therefore provided the `Camera` service. In the first experiment, the face detection service was available on a single iPAQ. If a face was found in the picture, the response time of the program was 75 seconds. If no face was detected, the response time was 159 seconds. The face detection code uses an image pyramid that is exhaustively searched. Once a face is found, the face detector terminates. This explains why the detector takes longer if no face can be found. In the second experiment, we ran the single face detection service on the 800MHz laptop. Depending on whether a face was found or not, the overall response time was 70 and 72 seconds, respectively. This example shows that it may be useful to allow a spatial view iterator to return after a finite number of virtual nodes have been visited. For instance, if the face detection service is replicated in the network, the nested iterator will try to find each face detector and apply it to the picture `p`. Clearly, this is redundant work unless the detectors are of different qualities. We plan to extend the language to allow users to specify an upper bound on the number of visited nodes by a spatial view iterator. In the case of the discussed example, the upper bound of 1, which means visiting at most one detector, is a reasonable choice.

4.4 Augmented-Reality Gaming

Multi-player on-line gaming on the Internet has become increasingly popular and has developed into a profitable business, as well as new area of academic research. In fact, prototype systems have already been developed that support multi-player games on wireless ad-hoc networks with handheld devices. Figure 14 and Figure 15 show an augmented-reality (AR) Pacman game for ad-hoc networks of mobile devices. This game is a new version of the popular 1980's pacman game. Each player has a GPS enabled PDA, which presents an interface as in Figure 14, with virtual obstacles, pellets (food for pacman to eat), ghosts (played by opponent players), and pacman (the player). The player eat pellets by moving

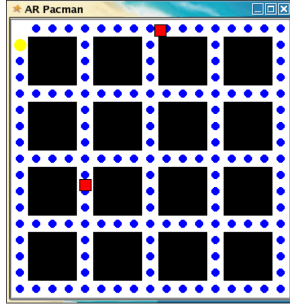


Figure 14. The AR Pacman game interface.

across their locations. The goal for the pacman is to win by eating all the pellets without being caught. He/she will lose if caught by a ghost. The Human Pacman project[11] developed a prototype system to play this game in a wide outdoor area.

The core of this game can easily be written in SpatialViews. Figure 15 shows a SpatialViews implementation. Each player runs this program on his/her PDA. The outer iterator propagates the program to all player nodes (PDAs). Every 4 seconds, the program initiates the collection of the current locations of all players, and updates the display. The inner iterator does the location collection. If the collection takes longer than 4 seconds (the time constraint), the results can be discarded. The `show` method in the `Pacman` interface renders a graphical interface on the PDA, showing the playground, obstacles, pellets, and all players' locations. Obstacles and pellets are supposed to be generated and maintained by the `show` method.

We did some first experiments for this location collection code with the help of 5 student participants. Each student carried an iPAQ handheld PC and a Garmin geko 201 GPS. The GPS device was connected to each iPAQ's serial port. The iPAQs communicated through 802.11b in ad-hoc mode. The students were playing the game in two parking lots of about 100m$\times$100m each.The experiments showed that an update rate of 4 seconds was feasible for our program and hardware. The GPS receivers provided an accuracy of 2.5 meters to 5.5 meters.

```
public interface Pacman {
  // Render a graphical interface including
  //  pacman, ghosts, peletts, and obstacles.
  public void show(Container c);

  // Get the role of the player: pacman or ghost.
  public Role getRole();
}
public class PacmanGame {
  public static void main(String[] args) {
    spatialview sv = Pacman @ SpaceDefs.PlayGround;
    visiteach x : sv every 4 forever {
      Container c = new Container();
      visiteach y : sv within 4 {
        Location l = System.currentLocation();
        c.add(new Player(y.getRole(),l);
      }
      x.show(c);
    }
  }
}
class Player {
  Location l;
  Role r;
  public Player(Role r, Location l)
    { this.r=r; this.l=l; }
}
```

Figure 15. An AR Pacman game program.

5. Discussion

In this section, we discuss miscellaneous issues in the design, implementation, and evaluation of SpatialViews.

5.1 New Language vs. Library

Effective programming for ad-hoc networks requires abstractions that do not exist in traditional models. These abstractions are needed to represent dynamic grouping, space, location and time resolution, discovery, routing, and in-network reduction. A library may provide a rapid implementation for a new programming model, because it does not require new tool chains or learning new language constructs. However, a new language is able to provide better support for the new programming model in terms of effectiveness of compiler-time analyses and opportunities for compiler optimizations. As a result, we expect compiler generated code to be more efficient than the corresponding program version based on a library (API) implementation of the programming model. New language abstracts are expressed explicitly, making the code more readable and maintainable. Finally, SpatialViews was also designed to serve as a vehicle to investigate different language features and compiler optimizations.

5.2 Units of Measurements

Units of measurements are currently not explicitly specified in SpatialViews. The current language assumes that length is always specified in meters and time always in seconds. This is a deficiency of the language that will be addressed in a future language release. The lack of units may initially lead to some confusion, but allowed the rapid development of our prototype with focus on the major design issues such as location-aware service discovery and quality of results. Explicit specification of units with enhanced language features, such as proposed in [2], will be investigated as part of a future language release.

5.3 High-Level Iteration Transformations

An interesting analogy can be made between Δs or Δt specification and traditional index-set splitting[3]. The target space in a spatial view definition can be thought of as the iteration space of a traditional loop. The geographic evenly distributed iteration has a similar flavor as traditional index-set splitting. Based on this observation, other traditional loop transformations may be applicable in the context of spatial view iterations. An example is loop flattening. Using a straightforward implementation, the code in Figure 16 requires one flooding of the network to find lighting sensors, followed by additional flooding to find cameras. If loop flattening is performed, only one flooding is necessary, which will collect both sensor readings and camera images, and select the correct results in the end. Preliminary experiments with one laptop computer and three HP iPAQs showed that the transformed version using loop flattening may run up to five times faster than the original version. Other loop based compiler optimizations such as loop interchange and loop fusion are currently under investigation.

5.4 Security and Privacy

Security and privacy are important issues in ad-hoc networks, where mostly unidentified nodes join and leave transparently. These issues become more important in a network running mobile code such as SpatialViews programs. Security and privacy are important for the migrating program as well as the participating nodes in

```
Container c = new Container();
spatialview SensorView = LightingSensor @ CampusB;
visiteach s : SensorView {
  if (s.read()>0.5) {
    Location loc = System.currentLocation();
    spatialview CameraView = Camera @ new Circle(loc,5);
    visiteach cam : CameraView
      c.addElement(cam.getPicture());
  }
}
```

Figure 16. An example program for loop flattening.

the network: A migrating program carrying user data needs to be protected from a malicious node, and a node needs to be protected from malicious migrating code. These issues are challenging not only for the language design, but also for the design of the whole system involving almost all other layers, including runtime library, virtual machines, operating systems, and hardware. Our current working assumption is that SpatialViews applications are going to be used in either a network of trusted members or on top of a trustworthy virtual machine or operating system that provides the protection needed. Providing security in a SmartMessages virtual machine is currently under investigation[48].

6. Related Work

Programming of ad-hoc networks has become a research focus in the past decade. Jini[43] is an architecture that supports service discovery and spontaneous networking. SpatialViews shares with Jini the same approach of naming services with Java interfaces. However, in contrast to Jini and other service discovery architectures[1], SpatialViews does not assume the existence of network-wide lookup services or service directories.

In recent years, programmability of sensor networks has become a hot research area[5, 47, 44, 14, 17, 28, 30, 7]. TinyOS[17] and nesC[14] provide a component-based event-driven programming environment for Berkeley Motes. nesC is an extension to C that supports and reflects TinyOS's design. TinyOS and nesC use Active Messages, which is similar to program migration in SpatialViews, but uses non-migrating handlers instead of migrating code. Maté[28] is a tiny virtual machine built over TinyOS for sensor networks. It allows capsules in bytecode to forward themselves through a network with a single instruction, which enables on-line software upgrading for large-scale sensor networks. Impala[30] also provides an event-based programming model, and emphasizes issues such as on-line software updates and adaptability. SensorWare[7] provides lightweight mobile scripts for sensor networks and is very similar to SmartMessages. Hood[47] and Abstract Regions[44] provide similar abstractions as SpatialViews, i.e., grouping nodes based on their properties. Blum et al.[5] proposed the concept of entity for an addressable group of sensors that monitor an event. TAG[31] considers a sensor network as a database, and provides a high-level SQL-like language to query it. Location is one property of the database about which queries can be made.

Programming models for massive networks of tiny embedded systems have also been studied. Such a network may contain millions of nodes, each of which is as small as a grain of sand. Nagpal presented a high-level language to program a sheet of agents similar to epithelial cells to form a global-specified shape just through local computation and communication[33]. Butera designed a programming model for paintable computers, which are small enough to mix with paint[9]. The major abstraction is process fragments migrating among nodes as the basic elements of a self-assembly process.

Migratory execution as seen in SpatialViews and implemented by SmartMessages has been extensively studied in the literature, especially in the context of mobile agents[10, 46, 27, 16, 15, 45]. Unlike a typical mobile agent system which makes migration a programming primitive, SpatialViews hides it in an iteration of virtual nodes named by properties.

Ad-hoc networking has been extensively studied[38, 21, 8]. Iteration in SpatialViews were implemented based on the same basic ideas of those ad-hoc network routing algorithms. In particular, it is not novel to use georgraphic information for addressing and routing. Navas and Imielinski proposed GeoCast for both geographic unicast and geographic multicast over the Internet[34]. Ko and Vaidya improved GeoCast for mobile ad-hoc networks[25]. Karp et al. proposed perimeter forwarding to recover from local maximal in greedy routing using node locations[23]. Li et al. proposed GLS, a location database that uses of geographic hierarchy to serve location queries with a server close to the querier in geographic routing[29]. And there have been also works for geographic multicast after GeoCast. Huang et al. proposed mobicast to disseminate packets into a moving and changing delivery zone[18]. Compared to those works, the geographic iteration in SpatialViews is different in that the expected node density can be specified for the target region, allowing redundant nodes to be avoided.

SpatialViews deals with time constraint. However, the time constraint in SpatialViews is significantly different from previous systems with strict time constraints, e.g. the time constraints in the Time Warp OS[20]. In Time Warp OS, the messages generated by a parallel discrete event simulation system have to be received in a nondecreasing timestamp order. This restriction can never be violated in order to guarantee the correctness of the simulation. Time Warp OS uses a *process rollback* mechanism to implement the time constraints. In contrast, the time constraint in SpatialViews is a soft deadline which should be better described as a budget. It is the amount of time that a programmer is willing to spend to finish a spatial view iteration. If the program spends more than the budget, further iteration will be prevented, but no rollback is necessary if ever possible. The time constraint in SpatialViews is one way for the programmer to tune the trade-off between the iteration time and the quality of results.

7. Conclusions and Future Work

Ad-hoc networks are an exciting new target platform with a wide open application potential. SpatialViews is a simple yet expressive high-level programming language for ad-hoc networks. The language tries to hide enough details about the underlying volatile target system while giving a programmer sufficient control over the efficiency and resource usage of the program as well as the quality of the computed result. A wide range of applications can be implemented in SpatialViews, indicating the expressiveness of the language. This paper discusses four applications together with possible optimizations. Simulation results and physical measurements showed the efficiency of the compiler generated code. The language can also serve as a testbed to investigate different compiler and runtime optimizations, and allows insights into the characteristics and requirements of programs executing on a volative, dynamic, and heterogeneous network.

Future challenges include compiler and runtime optimizations that take advantage of particular network characteristics such as network topology, degree of dynamic behavior, and node and communcation failure rates. Investigating the mathematical foundations of best-effort semantics is another important future challenge. It is not clear whether approaches used to specify nondeterministic languages can be extended to incorporate a best-effort model. A prototype version of our SpatialViews compiler, run-

time system, and debugging/visualization environment is available at http://www.cs.rutgers.edu/spatialviews.

Acknowledgment

We would like to thank Marios Dikaiakos and the anonymous reviewers for their insightful comments.

References

[1] W. Adjie-Winoto, E. Schwartz, H. Balakrishnan, and J. Lilley. The design and implementation of an intentional naming system. In *SOSP*, 1999.

[2] Eric Allen, David Chase, Victor Luchangco, Jan-Willem Maessen, and Guy L. Steele Jr. Object-oriented units of measurement. In *OOPSLA*, Vancouver, British Columbia, Canada, October 2004.

[3] Randy Allen and Ken Kennedy. *Optimizing compilers for modern architectures*. Morgan Kaufmann, 2001.

[4] T.E. Anderson, D. E. Culler, and D. A. Patterson. A case for networks of workstations: NOW. *IEEE Micro*, 15(1):54–64, February 1995.

[5] Brian Blum, Prashant Nagaraddi, Anthony Wood, Tarek Abdelzaher, Sang Son, and Jack Stankovic. An entity maintenance and connection service for sensor networks. In *MobiSys*, 2003.

[6] C. Borcea, C. Intanagonwiwat, P. Kang, U. Kremer, and L. Iftode. Spatial programming using Smart Messages: Design and implementation. In *International Conference on Distributed Computing Systems (ICDCS'04)*, Tokyo, Japan, March 2004.

[7] A. Boulis, C. Han, and Mani Srivastava. Design and implementation of a framework for efficient and programmable sensor networks. In *MobiSys*, 2003.

[8] Josh Broch, David A. Maltz, David B. Johnson, Yih-Chun Hu, and Jorjeta Jetcheva. A performance comparison of multi-hop wireless ad hoc network routing protocols. In *MobiCom*, 1998.

[9] William J. Butera. *Programming a paintable computer*. PhD thesis, MIT, February 2002.

[10] Luca Cardelli. A language with distributed scope. In *POPL*, 1995.

[11] Adrian David Cheok, Siew Wan Fong, Kok Hwee Goh, Xubo Yang, Wei Liu, and Farzam Farbiz. Human pacman:a mobile entertainment system with ubiquitous computing and tangible interaction over a wide outdoor area. In *Fifth International Symposium on Human Computer Interaction with Mobile Devices and Services*, 2003.

[12] S. Dashtinezhad, T. Nadeem, B. Dorohonceanu, C. Borcea, P. Kang, and L. Iftode. TrafficView: A driver assistant device for traffic monitoring based on car-to-car communication. In *IEEE Semiannual Vehicular Technology*, Milan, Italy, May 2004.

[13] eDonkey. *homepage*. http://www.edonkey2000.com.

[14] David Gay, Phil Levis, Robert von Behren, Matt Welsh, Eric Brewer, and David Culler. The nesC language: A holistic approach to networked embedded systems. In *PLDI*, 2003.

[15] Robert S. Gray. *Agent Tcl: A flexible and secure mobile-agent system*. PhD thesis, Dartmouth College, June 1997.

[16] Robert S. Gray, George Cybenko, David Kotz, Ronald A. Peterson, and Daniela Rus. D'Agents: Applications and performance of a mobile-agent system. *Software: Practice and Experience*, May 2002.

[17] Jason Hill, Robert Szewczyk, Alec Woo, Seth Hollar, David Culler, and Kristofer Pister. System architecture directions for network sensors. In *ASPLOS*, 2000.

[18] Qingfeng Huang, Chenyang Lu, and Gruia-Catalin Roman. Spatiotemporal multicast in sensor networks. In *SenSys*, 2003.

[19] The IETF Mobile Ad-hoc Networks (manet) Working Group. *homepage*. http://www.ietf.org/html.charters/manet-charter.html.

[20] D. Jefferson, B. Beckman, F. Wieland, L. Blume, M. DiLoreto, P. Hontalas, P. Laroche, K. Sturdevant, J. Tupman, V. Warren, J. Wedel, H. Younger, and S. Bellenot. Distributed simulation and time warp operating systems. In *SOSP*, 1987.

[21] David B Johnson and David A Maltz. Dynamic source routing in ad hoc wireless networks. In Imielinski and Korth, editors, *Mobile Computing*, volume 353. Kluwer Academic Publishers, 1996.

[22] P. Kang, C. Borcea, G. Xu, A. Saxena, U. Kremer, and L. Iftode. Smart messages: A distributed computing platform for networks of embedded systems. *The Computer Journal, Special Issue on Mobile and Pervasive Computing*, 47(4), January 2004.

[23] Brad Karp and H.T. Kung. GPSR: Greedy perimeter stateless routing for wireless networks. In *MobiCom*, 2000.

[24] Kazaa. *homepage*. http://www.kazaa.com.

[25] Young-Bae Ko and Nitin H. Vaidya. Location-aided routing (lar) in mobile ad hoc networks. In *MobiCom*, 1998.

[26] U. Kremer, Y. Ni, and A. Stere. Spatial Views language specification, version 1.0. Technical Report DCS-TR-563, Department of Computer Science, Rutgers University, November 2004.

[27] Danny B. Lange and Mitsuru Ishima. *Programming and deploying Java mobile agents with Aglets*. Addison-Wesley, 1998.

[28] Philip Levis and David Culler. Maté: A tiny virtual machine for sensor networks. In *ASPLOS*, 2002.

[29] Jinyang Li, John Jannotti, Douglas S. J. De Couto, David R. Karger, and Robert Morris. A scalable location service for geographic ad hoc routing. In *MobiCom*, 2000.

[30] Ting Liu and Margaret Martonosi. Impala: a middleware system for managing autononmic parallel sensor systems. In *PPoPP*, 2003.

[31] S. Madden, M. J. Franklin, J. Hellerstein, and W. Hong. TAG:a Tiny AGregation service for ad-hoc sensor networks. In *OSDI*, 2002.

[32] Mutella. *homepage*. http://mutella.sourceforge.net.

[33] Radhika Nagpal. Programmable self-assembly using biologically-inspired multiagent control. In *AAMAS*, Bologna, Italy, July 2002.

[34] J.C. Navas and T. Imielinski. geocast–geographic addressing and routing. In *MobiCom*, 1997.

[35] Yang Ni, Ulrich Kremer, and Liviu Iftode. Spatial Views: Space-aware programming for networks of embedded systems. In *The 16th International Workshop on Languages and Compilers for Parallel Computing (LCPC 2003)*, October 2003.

[36] Yang Ni, Ulrich Kremer, and Liviu Iftode. A programming language for ad-hoc networks of mobile devices. In *The 7th Workshop on Languages, Compilers, and Run-time Support for Scalable Systems (LCR 2004)*, Houston, TX, October 2004.

[37] Pac-Manhattan. *homepage*. http://pacmanhattan.com.

[38] Charles. E. Perkins. *Ad hoc networking*. Addison-Wesley, 2001.

[39] Nissanka B. Priyantha, Allen K. L. Miu, Hari Balakrishnan, and Seth J. Teller. The cricket compass for context-aware mobile applications. In *MobiCom*, 2001.

[40] Hanan Samet. *The Design and Analysis of Spatial Data Structures*. Addison-Wesley, Reading, MA, 1990.

[41] Sun Microsystems, Inc. *Java 2 Platform, Micro Edition (J2ME)*. http://java.sun.com/j2me.

[42] Sun Microsystems, Inc. *The K virtual machine*. a white paper available at http://java.sun.com/products/cldc/wp/.

[43] Jim Waldo. The Jini architecture for network-centric computing. *ACM Communications*, July 1999.

[44] Matt Welsh and Geoff Mainland. Programming sensor networks using abstract regions. In *NSDI 2004*, March 2004.

[45] D. Wetheral. Lessons from a Capsule-based system. In *SOSP*, 1999.

[46] James E. White. *Telescript technology: mobile agents*, 1996. General Magic, Inc. White Paper.

[47] Kamin Whitehouse, Cory Sharp, Eric Brewer, and David Culler. Hood: A neighborhood abstraction for sensor networks. In *Mobisys 2004*, June 2004.

[48] Gang Xu, Cristian Borcea, and Liviu Iftode. Toward a security architecture for smart messages: Challenges, solutions, and open issues. In *Proceedings of the First International Workshop on Mobile Distributed Computing*, May 2003.

[49] P. Zhou, T. Nadeem, P. Kang, C. Borcea, and L. Iftode. EZCab: A cab booking application using short-range wireless communication. In *Proceedings of the 3rd IEEE International Conference on Pervasive Computing and Communications (PerCom)*, March 2005.