ITR: A Framework for Environment-Aware, Massively Distributed Computing

David Evans (PI), Tarek Abdelzaher, David Brogan
University of Virginia, Department of Computer Science
Submitted 13 November 2001

Project Summary

The computing environments of the future will involve large numbers of small and inexpensive devices, communicating with each other and interacting with their physical surroundings to accomplish a task. We propose to develop the programming languages, tools and methods necessary to produce robust, scalable and resource-efficient programs for these new computing environments.

These new computing environments will require a fundamentally new computing paradigm that will differ in both substance and scale from previous work on concurrent programming and distributed systems. Unlike current processors, processors in future systems will have direct links to the physical environment through distributed sensors and actuators. Using thousands of tiny computational elements in place of a smaller number of larger computing devices enables a non-obtrusive computing medium with finer granularity of sensing and actuation. Computing capacity can thus be embedded seamlessly into physical objects, for example, by spraying a target spot with a dust of computational elements for data measurement, embedding computing motes into fabric, equipment casing, ink, or paint, or dropping a myriad of small computationally augmented sensors on a disaster area to search for survivors and determine structural damage.

While the hardware that enables this is becoming a reality, development of the software faces considerable challenges. These challenges arise from scale, unreliable components, dynamic system topology, massive interaction with the physical environment in real-time, and the need to reason about emerging aggregate properties as opposed to individual component behavior. In this research we propose to develop theory, methods and tools for massively distributed, environment-aware computing (more succinctly referred to as *swarm computing*).

The state of swarm computing today is similar to that of sequential computing in the early 1950s. Developers painstakingly produce swarm programs by designing and programming the actions of individual devices, and converge on an acceptable program through extensive simulation and experimentation. In the pre-compiler era, skeptical programmers believed that a mechanical process could not possibly produce code of comparable quality to that produced by highly skilled machine coders and that the cost of machine time is high enough to outweigh any possible savings in programmer effort. The state of swarm programming today is similar: devices are still expensive enough and resources limited enough that applications are constructed in a painstaking, trial-and-error way to produce acceptable results for a given problem. This will change soon as the costs of devices decreases and their capabilities increase, the human time required to develop swarm programs will come to dominate the other costs.

Hence, we propose to develop the methods, languages and tools that will enable the principled and efficient development of swarm programs. A key contribution will be a programming language and development tools that enable programmers to construct and reason about programs in these environments at an appropriate level of abstraction. The language must allow the programmer to express programs at the level of aggregate behavior rather than the behavior of individual devices. Further, it should allow developers to control resource consumption at levels of detail appropriate to their requirements and application. Instead of hand coding programs for a particular “machine” (in this case, a collection of devices operating in a physical environment), we will provide programmers with the ability to write a program once and automatically adapt it to different deployments. Instead of requiring trial-and-error experiments to determine how a swarm program will behave in a new environment, we seek to develop principles and tools for predicting behavior.

ITR: A Framework for Environment-Aware, Massively Distributed Computing

Advances in processor, memory and radio technology have made it possible to build small and cheap devices that are capable of sensing and altering their environment, communicating with nearby devices using short-range wireless channels, storing a small amount of state, and performing a limited amount of computation. We will be able to deploy thousands to millions of these devices to perform a wide range of applications such as monitoring poorly accessible or dangerous environments like disaster areas, hostile territories and active nuclear fields. Hostile or dangerous terrains make it impossible to build fixed infrastructures of powerful and expensive hosts. Instead, swarm computing advocates the use of myriads of inexpensive devices placed arbitrarily in the environment and left largely unattended. The capabilities of the swarm are drawn from network scale rather than from powerful individual components resulting in a high degree of built-in fault tolerance whereby the failure of many devices causes only a marginal decrease in the effectiveness of the application.

We propose to develop a programming language for swarms of devices that allows a programmer to express an application in terms of desired aggregate behavior rather than explicitly programming individual devices. We will produce a *program synthesizer* that automatically generates the appropriate device programs that will implement the aggregate behavior for a particular deployment. The program synthesizer draws from a library of primitives that can be combined to produce complex applications with known scaling and non-functional properties. In addition to analytical results, we will test our approach using simulations and using a physical swarm composed of wireless devices with sensors and actuators.

Our multidisciplinary team combines expertise in many areas, and we feel it is necessary to combine different skills and experiences to tackle this problem. Tarek Abdelzaher focuses on networks and control theoretic approaches to systems problems; David Brogan specializes in multiagent systems and simulation; David Evans (PI) researches programming languages, software engineering and security.

1. Scientific Objectives

The design of programming languages for computing swarms poses several new challenges that stem from three interdependent peculiarities of these networks: *scale*, *physical embedding*, and *device-administrator ratio*.

Scale. Swarm computing involves, as a main feature, a massive number of devices. We therefore must derive system properties that emerge as the number of devices increases. These properties need not necessarily hold for smaller numbers of devices. An important evaluation criterion of our language implementation would be the minimum number (or density) of devices that needs to exist for the programmed properties to hold. We call that number, *critical mass*. The critical mass limitation is, in a way, the inverse of scalability limitations. Traditional distributed computing typically designs algorithms that scale as the number of components increase. The scalability of a distributed algorithm is defined by the maximum number of components for which performance is acceptable. In contrast, primitives of swarm programming languages will have a distributed implementation that requires a minimum critical mass. The designers of swarm applications need not be concerned with providing the desired behavior when the number of devices is low.

Probability theory provides a rich vehicle for formal analysis of such scalable algorithms. Properties of stochastic processes typically approach deterministic values when the number of trials increases. Similarly, a swarm application will elicit behaviors that emerge and stabilize with scale. For example, implementing filters slightly predisposed for propagation of higher priority traffic in the network, will result in a strict and predictable prioritization scheme when the number of hops is large enough. Note that typically, a trade-off exists between algorithm complexity and critical mass. In the prioritization example, it may be possible to enforce strict priorities on a per hop basis. Such strict per-hop algorithms have a minimum critical mass of one. However, they are difficult to implement due to distributed contention over the shared medium from multiple sources carrying traffic of different priorities. Implementing an algorithm that simply has a higher probability of sending higher priority traffic first across a shared medium is easier but requires a larger path (i.e. a larger critical mass) for strict prioritization to emerge end-to-end. Hence, our solutions will generally tend to rely on the simplest possible per-node behavior that is analytically determined to be likely (probabilistically) to provide desired global properties when critical mass is observed. We may also implement different flavors of algorithms that differ in their simplicity/critical-mass trade-off. For example, as the system ages to the point when the number of active nodes decreases below the critical mass of the simplest algorithm, a more computationally intensive algorithm can be invoked to maintain the property under the new resource constraints.

Physical Embedding. Swarm computing devices are typically equipped with sensors and actuators that make them capable of interacting directly with a physical environment. This capability sets swarms apart from traditional networking infrastructures (such as the Internet), where the main problem is to control the flow of information. Algorithms will need to be developed for controlling the exchange of information between the swarm computing network and the environment, as well implementing distributed actions on the environment when needed.

Physical embedding also permits a different programming paradigm. Current programming languages, at their very core, are vehicles for manipulating abstractions of computer hardware. For example, data structures and operations performed on them are abstractions for manipulating content of memory locations and registers. In contrast, in swarm computing the manipulated medium extends beyond computing hardware to encompass the physical environment as well. Objects in that physical environment might have attributes that one may read within a swarm program. This data, depending on its different attributes, may initiate events that manipulate the state of an object (e.g., by eliminating these objects from the environment as in a battle scenario), or change object attributes via nearby actuators. Our programming system should be rich enough to allow programming physical interactions with the environment the same way we program manipulation of logical data structures today. We will not focus on the physical mechanisms for affecting and sensing the environment, but rather on developing manipulation and observation primitives with semantics suitable to what is possible physically and on developing algorithms that exploit shared state using physical environments.

Device-Administrator Ratio. In typical environments today, there are a small number of computers per system administrator. Most individuals have to act as their own system administrator for a single machine. In larger organizations, a single overworked system administrator may be responsible for hundreds of machines. With swarm computing, we need to have thousands or millions of computing devices per administrator. We cannot expect people to repair or manipulate individual devices. Hence, swarm computing applications need to be able to operate largely autonomously, even in the presence of device failures and environmental changes.

Due to their unattended mode of operation, swarm devices need more attention to be given to their automatic configuration, group formation, failure detection, and fault-tolerance. Swarm applications cannot expect human administrators to repair malfunctioning devices; hence swarm programs should not rely on identifying individual devices. This implies that most algorithms would have to support stateless operation or store state in a distributed, environmental fashion.

Summary. The primary scientific objectives of the proposed work are:

1. Develop a theory, techniques and tools for reasoning about the scalability of massively distributed programs.

2. Develop techniques for analyzing and synthesizing swarm programs constructed by combining primitives defined over interacting groups.
3. Define a class of primitives with known scaling properties and methods and languages for describing their functional and non-functional properties.

2. Background

Advances in hardware have reduced the cost of computation and communication to the point where it is now reasonable to imagine deploying applications on thousands to millions of independent computing and communicating devices with sensors and actuators. In this section, we summarize recent trends in hardware devices and survey related work in programming paradigms, networks, and multiagent systems relevant to the proposed work.

Devices. The steady improvement of microprocessors has lead to devices costing a few dollars that are as capable as desktop computers of 1985. Recent advances have led to dramatic reductions in the cost, size and power consumption requirements for electronic devices, wireless communications, and microelectomechanical systems (MEMS). Several hardware models of sensor nodes have been developed, primary among them being the Rockwell WINS nodes [17] and the Berkeley motes [16]. The Smart Dust project at Berkeley is building cubic millimeter-scale self-powered autonomous devices with the ability to perform computation and communication [60]. These devices currently run on a 4 MHz microcontroller and have 8 KB of program memory and 512 bytes of data memory. Versions of these devices are being developed that use infrared, radio frequencies, and lasers to communicate and have sensors for temperature, brightness, and magnetic fields. Current versions of the devices last for about one week with continuous operation. Eventually, we may be able to construct even smaller and cheaper devices using molecules. It is expected that prototype molecular electronic devices will be built within a few years, although simple computers are probably at least a decade away [59]. The age of inexpensive computing and communicating devices is on the horizon. We can build the devices – the question is whether or not we can program them to do useful things.

Programming Paradigms. Work on amorphous computing has produced promising results in programming paradigms for massively distributed systems [95]. The amorphous computed is a similar application domain to swarms, except the devices are generally less powerful and considered to be immobile. One technique for programming an amorphous computing medium is the Growing Point Language (GPL) [94]. GPL uses a botanical metaphor of growing points that can secrete pheromones that act as messages to nearby nodes, and lay down material that alters the state of particles in the amorphous computing medium. Coore has demonstrated that this model is sufficiently powerful to program complex structures such as the interconnect topology of a CMOS circuit. Our emphasis will be on techniques for describing and reasoning about the behavior of swarm programs.

Tuple spaces are a model for concurrent programming that obviate the need for devices to have individual addresses [63, 91, 92]. A tuple space is a global shared associative memory. It provides a method for adding, deleting, and reading information. Since values in the tuple space cannot be modified, there is no need for synchronization – the values themselves provide all the necessary coordination. The tuple space approach has many desirable properties, but a number of challenges would need to be overcome before it could be used in for swarm computing. Scaling and mobility present major problems, since it is normally assumed that all nodes can see the entire tuple space. Partitioning a tuple space into neighborhoods may offer a partial solution, but this requires substantial changes to the programming model. One approach is explored by LIME [93], a tuple space system designed for mobile environments. In LIME, tuple spaces are transiently shared and distributed across mobile hosts according to network proximity. In a swarm computing environment, it will not be practical to maintain a traditional tuple space even if it is transient. Instead, there may be a use for an "ether" space where tuples are broadcast to neighboring nodes. It is unclear whether or not the much weaker semantics can support a useful programming model, but we believe it will be worthwhile to explore weaker variations on tuple space semantics for a swarm programming.

Networking. Although the principles of swarm computing are not necessarily tied to wireless networks, most interesting swarm computing environments are wireless. Chemical diffusion and light are two wireless communication methods, but the most promising short-term technology is radio. Efforts to develop IEEE 802.11b [61] for high-bandwidth wireless local-area networks and the Bluetooth [62] standard for low-cost, low-power small area networking will improve standardized, low-power wireless networking protocols. Nagpal [5] and Bulusu and Heidemann et al. [6, 12] have studied location estimation using GPS beacons. The Amorphous Computing group at MIT has studied the problem of organizing a global coordinate system from local information [64]. Geographical forwarding is a promising dynamic addressing protocol, that fails in sparse regions of the network. Xu, Heidemann, Estrin [6, 12] have proposed a geographical adaptive fidelity (GAF) algorithm in which equivalence classes of nodes are formed from a routing perspective. The Intentional Naming System (INS) is a resource discovery and service location system designed to support dynamic networks [88]. This system requires that devices periodically advertise their name specifiers to a centralized server, which manages the communication among the group. A swarm implementation of INS requires eliminating the server, but alternative device-level message-propagation protocols may suffice. The arbitrary structures of the network and the limited computational abilities of the nodes make developing network primitives for swarm computing a challenging research area.

Instead of routing messages between specific devices, work in sensor networks has explored protocols that transmit information to devices as needed based on application constraints. A typical scenario is a collection of sensors is dropped on a battlefield to monitor troop movements and communicate relevant findings back to a command center. One technique is a communication paradigm known as *directed diffusion* that distributes messages in a data-centric way [90]. Sensor devices produce data in the form of attribute-value pairs. A sensing task is disseminated through the sensor network by setting up gradients that draw data matching requested attributes towards the request origin. Rather than specify this device directly, however, directed diffusion sends data back in the direction of its request without needing to route data to a specific origin device. The use of gradients enables robust, power-efficient communication without the need for complex routing algorithms. In addition, these techniques take advantage of task-specific algorithms for aggregating sensor data. Subramanian and Katz have generalized the techniques to develop an architecture for building self-configurable systems where a large number of sensors automatically coordinate to achieve a sensing task [89]. Swarm computing will use known techniques for organizing devices to produce a network suitable for a given program. One goal of this research will be to produce methods for self-organizing devices based on the desired properties of the application, instead of requiring explicit programming.

Multiagent Systems. Biological mechanisms used by insect swarms provide a source of inspiration for swarm computing. The behaviors of ants, bees, and fish demonstrate a variety of group behaviors that have been studied extensively over the past century. The distributed behaviors of ants produce self-organizing shortest-path selection [65, 66]. A novel approach to routing is inspired by the behavior ant colonies use to find the best route to a food source [84]. Insights are also gained from beetle circle defenses [86] and honeybees searching for a new home [85], foraging for food [67], and growing their hives [68]. Animals also demonstrate that grouping minimizes losses in hostile environments [69, 70]. We are inspired by biological systems, but don't believe attempting to duplicate them is the best way to produce swarm programs. As with humanity's attempts to fly by mimicking birds, attempts to copy biology directly failed; it was only when the underlying principles behind the behavior were understood that we could produce flying behavior with machines.

Some artificial intelligence researchers and roboticists have focused on building and simulating robotic systems that utilize these group behaviors. Research by Arkin, Mataric, Stone, and others [71, 72, 73] demonstrates that hierarchical combinations of simple behavioral primitives result in coordinated foraging, search and rescue, and soccer playing. These research projects explore the influence of heterogeneity, communication protocols, and uncertainty on task performance.

Computer graphics researchers have also made contributions to simulation of groups and distributed control algorithms. Reynolds accomplished early progress in the simulation of group behaviors [74] by developing simulated flocks of birds. Each bird uses only information about nearby neighbors and the simple algorithms fail to work if global knowledge is used. The group animation systems developed by computer graphics researchers Blumberg, Terzopoulos, and Thalmann [75, 76, 77] further demonstrate how simple behaviors can be combined to create more complex behaviors. In particular, these researchers have investigated the ways simple behaviors are blended and the way one behavior supercedes or truncates competing ones. Brogan's research with multiagent groups of simulated agents possessing significant dynamics investigates the ways physical limitations affect group behaviors [79, 80, 81, 82]. These systems also demonstrate that simulated groups with simple behaviors could adapt to a dynamic environment.

3. Research Plan

Our research plan is focused on creating the programming system shown in Figure 1. This comprises the following main tasks:

1. Designing *Seurat*,[1] a programming language for describing an application as a combination of primitive behaviors .
2. Building a library of primitive behaviors defined over groups.
3. Designing specification languages for the device and environment models. These languages need to be sufficiently detailed and precise to allow automated reasoning about the devices and environments they describe, but also abstract and flexible enough to allow reasoning about deployments that scale to millions of interacting devices.
4. Implementing development tools, compilers and deployment mechanisms for executing a program on a collection of small devices in a physical environment (the *program synthesizer*). The process of deploying a program on a computing swarm involves synthesizing the appropriate combination of primitive behaviors to implement the application on a given set of devices operating in a particular environment.

The main tools we have for these tasks are abstraction and hierarchy. Abstraction allows us to remove the details of physical devices and environments from the concern of the programmer; further, we are exploring new abstractions that provide convenient mechanisms for programmers to express interactions. As group size becomes large, a hierarchy of abstractions controls the propagation and inheritance of primitive behaviors in a systematic and analyzable way. The next four subsections describe each component or our architecture. Section 3.5 discusses our experimental platforms.

[1] Seurat is named for Georges Seurat, a pointillist painter. Pointillism is a painting technique where a painting is constructed by painting many small dots of color, and stronger hues are created by carefully placing dots of pure colors. Seurat is reported to have said, "They see poetry in what I have done. No. I apply my method, and that is all there is to it." We hope to achieve a similar effect with swarm programming – programmers will be able to construct what appear to be emergent programs by using well-defined and principled methods.

3.1 Seurat: A Programming Language for Swarm Computing

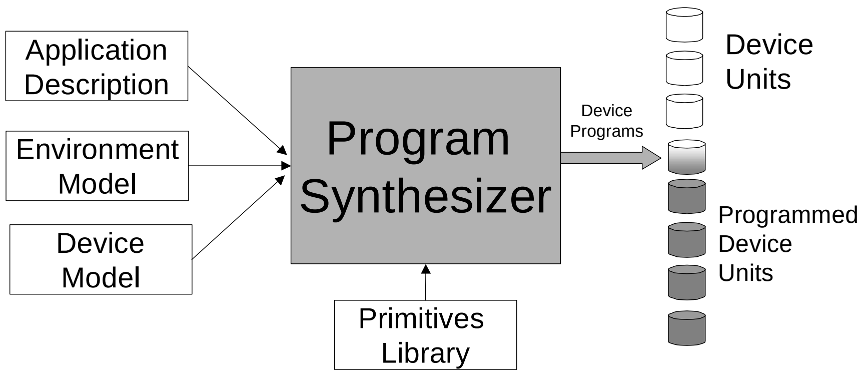

Figure 1. Swarm Programming System.

To build a swarm application, we need tools for creating individual device programs from a desired high-level behavior. We propose to design a program language that can be used to describe a desired aggregate behavior of an ad-hoc collection of devices. Just as pointillist artists paint points that represent shapes and styles, our programming synthesizer determines the device programs (the points) that accomplish the functional properties (the shapes) and the non-functional properties (the styles) desired by the programmer.

Aggregate Behavior Specification. A swarm program must specify the desired aggregate functional and non-functional properties of the application. Aggregate functional properties are expressed by combining primitives. In addition, primitives may be customized using both first order and higher order parameters. It is important to note, however, that Seurat does not allow behavior specification to program individual devices directly. This restriction is what enables the program synthesizer to produce scalable applications. For example, the swarm program functional description for a search and rescue application could be expressed as[2]:

```
  (disperse (scan-distance) |? scan (target-code)) > broadcast (found-signal)
 | listen (found-signal) >^ homing (found-signal)
```

The disperse primitive moves the swarm towards a state where no two devices are within its distance parameter of each other (in this case, a value that depends on how large a region the devices can scan). This is combined with the scan primitive, parameterized with a function that identifies the target to locate. The |? combinator indicates that disperse and scan should be done concurrently, until scan is satisfied. After this, the broadcast primitive takes over, transmitting a found-signal throughout the swarm to attract other devices to the location. Simultaneously, the listen primitive directs devices to listen for the found-signal. If it is heard, the device switches to the homing primitive that directs the devices to the signal location. The >^ indicates that when listen is satisfied, homing should preempt all other behaviors competing for the same resources (in this case, the actuators that move the device).

Non-Functional Properties. Note that the functional description does not provide any guidance as to *how* to do the primitive (disperse, broadcast, etc.) behaviors. Unlike a traditional programming language where there is typically only one way to perform a primitive operation determined by the

[2] This example is not meant to reflect the actual syntax and semantics of Seurat, but rather to give a flavor for what that language might be and an idea of what our vision is for programming by combining primitives defined as aggregate behaviors. Actually figuring out the details of the syntax and semantics is a challenging part of the proposed research. The rough syntax and semantics used here has many flaws and ambiguities, which would need to be addressed by a final language definition.

operand types, there are many possible ways of implementing each of the primitive swarm behaviors. There is no single best implementation, but rather a set of implementations that exhibit different non-functional properties over different scales and types of devices. The program synthesizer must choose a combination of these implementations that best satisfies the non-functional requirements of the applications and the properties of the devices and environment on which it will be deployed.

Non-functional properties include resource consumption, efficiency, security and survivability. The programming language needs to provide constructs for expressing these properties at varying levels of detail. For example, a home application for cleaning floors would need to specify the resource consumption requirements to indicate that it is important that device batteries not need to be recharged frequently. On the other hand, a search and rescue application for a hostile environment is more concerned with efficiency, security and survivability. Although these applications are functionally quite similar, their implementations are likely to differ considerably. The programmer, however, should not need to be concerned with those details. Instead, she should be able to separate the functional and non-functional requirements of the application and define different non-functional requirements at different levels of abstraction depending on her particular application requirements.

Aggregate Behavior Control. A key mechanism for realizing different non-functional properties lies in our ability to generate primitive implementations that trade-off resources for performance in a way that is consistent with the non-functional properties required. This trade-off should be guided by an objective function that expresses global performance requirements of the implementation. The device programs should be constructed to achieve these global performance requirements. To achieve this end, we break the problem into two independent questions: how to relate local performance of the device to global performance metrics and how to control global performance to adhere to specifications.

To provide an analytic foundation for answering these questions, we propose to develop real-time control theory, a new branch of distributed computing that describes the mathematical properties of controlling massively distributed software system components to achieve desired non-functional performance assurances. This theory relates time, information content, communication constraints, and environmental stimuli to internal software state representative of language primitive goals and progress. The theory will provide guarantees on the proximity of actual primitive behavior to the specified behavior. Such guarantees translate into specifications on the transient response of performance control loops upon load changes.

We shall investigate alternative swarm representations for the purposes of performance control under the new theory. The main distinction of the new representations is that they describe swarm dynamics independent of an accurate load and resource requirements model. Thus, lack of prior estimates of load and resources will not affect the integrity of the model used for performance control.

The investigators have applied control theory successfully in several case studies involving distributed computing applications including performance guarantees in web servers [37,38], Internet server delay control [30], proxy cache relative hit ratio control [32], and microprocessor thermal management [26]. We have also had success with differential equation modeling of software systems. Unlike queuing models and real-time scheduling models, differential equation models describe system dynamics independently of the characterization of input load. For example, in [34] we describe a liquid flow model of a real-time system for the purposes of controlling its deadline miss ratio. In [31,33] we show how a commercial server is modeled by a differential equation to handle overload. In [35], we describe a self-profiling subsystem that derives system models automatically without external intervention. We shall extend these models to massively distributed systems.

Our preliminary results [30,31,32,33] show that PID controllers can be used effectively for enforcing resource allocation decisions in large software systems when system parameters (differential equation coefficients) are constant. Elements of adaptive and robust control theory will be applied to situations where system parameters are unknown or change at run-time. For example, deadline misses in real-time systems increase nonlinearly with increased resource utilization. We shall investigate adaptive feedback control and robust control techniques to account for system parameter uncertainty and variations. To control a system with multiple objectives, multivariable control schemes will be developed to implement device algorithms that achieve an aggregate behavior of choice.

Manipulating Physical Environments. A programming language typically includes an abstract address space in which a programmer can create and manipulate objects. These objects provide an abstract way to manipulate bits represented by machine hardware and storage systems. In contrast, Seurat assumes a distributed machine that includes not only the traditional hardware and memory local to a single device, but also a myriad of sensors and actuators that constitute an integral part of the architecture. Programmers should be able to use abstract programming constructs to manipulate and sense physical environments outside the device, as programmers today are able to use abstract programming constructs to manipulate and acquire values from machine registers and storage. The right abstractions for dealing with state in a physical environment, however, remain to be discovered. Unlike traditional computer hardware, state represented in physical environments is subject to physical processes such as diffusion and evaporation.

Time. Time is a variable of prime importance in any system that interacts directly with a physical environment. Our team includes expertise in adaptive real-time computing which was the PhD topic of one of the investigators [40]. Our goal is to develop programming primitives with predictable timing performance. Techniques for predicting and controlling timing behavior have been investigated at length in the real-time systems community. Our particular interest lies in extending these techniques to massively distributed systems. In particular, we are interested in developing theory that relates timing guarantees on primitive behavior to network load, such that bounding the latter will imply satisfaction of the former. Systems that behave unpredictably on a microscopic scale often tend to have more predictable aggregate properties. We shall develop algorithms that derive aggregate performance models and utilize them to provide timing guarantees. Examples include utilization bounds that guarantee meeting individual primitive deadlines as long as aggregate resource utilization remains below certain limits. In initial work, we derived such bounds for tasks with unknown arrival times [29]. These results will be useful in the implementation of programming primitives that must execute in bounded time.

3.2 Building a Primitives Library

Implementation of high-level language primitives will require the design of distributed algorithms that achieve the aggregate behavior specified by the primitive. The collection of such algorithms constitutes the primitives library to be used by the compiler for code generation.

It is important to note that the primitives library is not a system library in the traditional sense – rather than being a fixed functional implementation, each primitive encompasses a large class of implementations that can be adapted and scaled to particular device, environment and behavioral requirements. Unlike the swarm programs that must be described at the level of aggregate behavior, implementations of library primitives define the behavior of individual devices. They are programmed using traditional programming methods (e.g., a Java program that uses an API which abstracts operations on device sensors and actuators). The challenge is to be able to analyze the primitives well enough to describe their behavior formally in a way that can be scaled to different numbers of devices and adapted to different devices and environments.

In an earlier section, we described the factors that underlie the most obvious differences between the nature of algorithms we shall develop for implementing swarm primitives and previous research in traditional distributed computing. We will provide primitives at various levels of abstraction. At the lowest level, we have primitives that deal specifically with communication; next, we have primitives for organization; at the highest level, we have behavioral primitives. To synthesize a swarm program, the program synthesizer must select different implementations of primitives at each level.

3.2.1. Communication Primitives

Communication support for swarm computing poses several challenges that arise from the three distinctive properties of swarm systems described earlier: scale, physical embedding, and administrator ratio. The challenges addressed in this project are as follows:

Addressing. When thousands of devices exist per administrator, the identity of a single device becomes irrelevant. Hence, new addressing techniques need to be developed which do not rely on single-device identifiers. Such schemes will rely on sets of diffused logical coordinates that create gradients along different geographic dimensions in the network.

Routing. Current routing algorithms rely on device identifiers that can be mapped into routing decisions such that a message is sent to the destination. In a network where individual devices do not have identifiers, routing may rely on other means such as physical clues from the environment or the gradients created by the addressing scheme. We call such clues and gradients, physical context. For example, geographic routing algorithms may be appropriate which forward messages to specific locations rather than specific devices. Geographic location, in this case, is an attribute of the physical context of the message destination. A calculus will be developed to allow developing routing algorithms that use addresses based on abstract coordinate systems for forwarding data towards the destination.

Transport layer services. Current networks rely on high-level communication abstractions such as sockets (IPC) in which a device can establish a data connection with another device for information exchange. In this model the device is permanently associated with the connection end-point. In contrast, when individual devices are not important, connection endpoints may be associated with pools of devices that share a particular context or attributes, such as all devices within one centimeter of location *x*. This pool of devices may dynamically change while preserving the end-point abstraction.

Congestion control. From the perspective of the global network of elementary nodes, a key problem is to ensure that the network is not too overloaded to perform its basic function – convey information. This function is achieved with congestion-control algorithms. The most prominent such algorithm is TCP congestion control. Unfortunately, TCP cannot work in the new environment because of its wireless, unreliable nature and because of the lack of an IP layer, or a similar layer that exports node addresses. A different form of congestion control will need to be implemented.

3.2.2. Organization Primitives

Organization primitives will provide higher-level functionality for group and hierarchy formation. To develop scalable behavioral primitives, we need primitives that scale by organizing devices into dynamic, adaptable hierarchies. The organization primitives must be designed to be resilient so that they succeed even when implemented by only a fraction of devices. Further, each primitive may be implemented in different ways that trade-off different non-functional properties. Example organization primitives include:

Team management. While individual devices are not important in the new computing environment, it will be important to create abstract higher-level entities that are addressable and can perform computational functions as dictated by the programmer. These entities will be implemented at a lower level by dynamically evolving teams of elementary nodes. Hence, team management will be an important challenge in the new environment.

Data management and storage. An important function of the network will be to collect and process data. This data will often need to remain in the network until it is explicitly requested, or until some critical event is identified that needs to be propagated immediately to the users. Hence, data management and storage will be an important network function. The problem is tightly related to that of web caching, except that in a sensor network case, data updates can originate from any node in the network. Moreover, no node is powerful enough to act as a centralized data cache or storage facility. Instead, fully distributed solutions are needed.

Priority control. The network may be employed to perform distributed tasks such as data communication and storage, team management, and others. In the presence of multiple tasks and limited resources the system should have a way of consistently prioritizing its operations. Protocols will need to be developed to determine priorities of different activities in a distributed manner and in a way that maximizes system utility. Mechanisms for priority enforcement will be implemented to carry out the priority order computed by the aforementioned algorithms.

3.2.3. Behavioral Primitives

The behavioral primitives are programmed using the system services as well as new device code. Devices provide different functionality, but all devices must provide standard interfaces that can be used to create the primitives and underlying services. The library must provide a set of behavioral primitives that provide sufficient functional behaviors to allow a large class of complex swarm applications to be written. One of the theoretical questions we will address is whether there is a small set of primitives from which a large well-defined class of applications can be created. In addition, the library must provide implementations of those primitives that can be used to provide different non-functional properties. One area we will explore is whether certain kinds of primitive implementations can be combined so that a few primitive implementations and combination mechanisms will provide access to a large design space of non-functional properties. Determining the set of behavioral primitives is part of the proposed research. Some examples of likely behavioral primitives include disperse, scan and homing introduced in Section 3.1. We expect that the final library will have around twenty behavioral primitives. We will define a class of swarm computing programs that can be created using the library primitives.

Here, we consider how disperse might be specified and implemented. The end state of disperse satisfies a constraint of the form that no two devices are within *d* distance of each other. A naïve implementation of disperse would program each device to detect the number of devices within *d* of it by transmitting a message with the appropriate power to cover the *d* distance region surrounding it. If one or more devices respond, move randomly for a short distance and try again. The naïve disperse implementation is neither time nor power efficient, but its simplicity and symmetry make it easier to reason about. Alternative implementations that offer different tradeoffs between time and power given different costs for communication, computation and movement will be explored. We have conducted preliminary experiments with an implementation in which devices communicate in a hierarchical way to locate areas of low density and then move towards those low density areas. This approach achieves dispersion in about 1/10th the time of the naïve implementation, but with significant communication costs. We have conducted preliminary experiments in which some fraction of the swarm runs the naïve implementation and the rest run the density driven implementation to illustrate that many points in the design space for trading off quick results and communication costs can be achieved with just two implementations.

3.3 Specifying Environments and Devices

To generate device implementations for a Seurat program, the program synthesizer needs to be aware of the application requirements, the properties of the devices in the swarm and the environment in which it will execute. The non-functional properties, capabilities and relative costs of communication, action and computation in the deployment environment determine the combination of primitive implementations that will be used on the devices. In addition, failure modes, the likelihood of hostile exogenous events and the importance of degrading gracefully for certain types of failures all affect the appropriate implementation choices. Our programming system is designed to separate those factors from the behavioral program by providing specifications of environments and devices and using the program synthesizer to select appropriate implementations.

A key challenge in the device and environment specifications is determining the appropriate level of abstraction. If the specifications contain too much detail, the complexity for both humans and the programming tools will quickly become unmanageable. If they contain too little, there will not be sufficient information available to the synthesizer to select the right combination of primitive implementations. We propose to take an approach based on variable levels of detail. For example, in some applications it may be necessary to describe the communication costs properties of the environment in great detail; in others, it may be sufficient to know a radius distance within which communication is almost always successful. Programmers should be able to use different levels of detail based on the requirements of their application, and rely on the programming system to make sensible assumptions in cases where details are not provided.

We also plan to explore methods for describing hostile environments, for example where an intruder will deliberately destroy clusters of devices or jam communications. Although not much work has been done in this area, our colleagues' work on describing and simulating vulnerabilities in Raptor [41] offers a good starting point for this exploration.

3.4 Deploying Swarm Programs

The Seurat programming model is different from the traditional compile, link, ship and execute sequence. Unlike in traditional systems where it is adequate to send identical binaries to all hosts, we anticipate most Seurat programs need to execute on a wide range of devices and in a variety of environments. The combination of primitives selected to implement a program depends heavily on the capabilities and number of available devices and the physical properties of the deployment environment. We envision a day when individuals would have a swarm deployment device in their home. It would (using initial discovery programs and devices) learn about the surrounding environment and be configured according to the number and kinds of devices its owner has purchased. A new Seurat program would be downloaded to provide some new functionality, and the program synthesizer would synthesize appropriate programs for the owner's devices and environment, install those programs on the devices and launch them to carry out the application. This all needs to happen without the owner doing anything other than perhaps selecting a new application to deploy.

Our short term goals are more modest. We assume device and environment specifications will be provided and focus on the problem of selecting the appropriate combination of primitive implementations for an application given its behavioral description, nonfunctional requirements, and the device and environment specifications.

Another crucial aspect of application development is debugging. Massively distributed programs are particularly difficult to debug; the nondeterminism and lack of reproducibility in the environment further exacerbate the problem. We will need to develop tools that allow programmers to monitor and adjust an application at a variety of levels. Occasionally, it will be useful to examine the state and operation of a single device. More often, though, debugging will need to be done at a higher level. Attempting to debug a swarm program by examining the state of individual devices would be like debugging a C++ program by examining the state of individual electrons in the machine core. Instead, we need to develop tools that allow programmers to monitor and evaluate properties of the swarm as a whole. Our preliminary work in this area has produced a tool that takes a simulation log and an evaluation function that produces a value for a snapshot of the world state and allows the programmer to see the value of that evaluation function over time [42]. This simple tool points the way to more adaptive and powerful techniques for debugging and evaluating swarm programs.

3.5 Experimental Platforms

We will conduct experiments using both physical and simulated environments. Physical experiments are useful for validating our approach and revealing important issues that may be hidden in a simulation. However, limits on the number of devices available to us in a physical experiment make it impossible to test scaling over a sufficiently large range. Hence, we will focus on simulation experiments, and use physical experiments to test the realism of our simulations.

Physical Experiments. At present, we have a lab with a limited number of Berkeley motes [16], which run a micro-threaded operating system called TinyOS [16]. Each mote has up to three embedded sensors. It includes a 8-bit 4 MHz micro-controller and has 8KB of program memory and 512 bytes of data memory. TinyOS, itself takes about 180 bytes of data memory, and around 4KB of program memory. Future versions of the motes, which we expect to get in January 2002, will have about 10 times the memory resources. The motes represent one possible point in the design space for swarm computing devices.

Simulator Experiments. We will conduct experiments using several different simulators augmented with analysis and code generation tools we develop. Because we need to deal with both low-level behavior of physical devices (e.g., modeling power consumption and communication reliability) and the high-level behavior of collections of millions of devices, there is no one simulator that is adequate for all aspects of the project. Our team currently has experience with several simulators spanning the design space we need to consider. These include:

Santa Fe Swarm simulator [43]. The Santa Fe Swarm simulator provides an easy interface for modeling device behaviors and decent tools for visualizing swarm programs. It scales to a few thousand devices, but is likely to be too slow for large scale experiments. In our preliminary work, we have used Swarm to simulate several implementations of swarm primitives. Its programming interface is simple enough that good undergraduate students are able to quickly develop and begin experimenting with primitive behaviors [44].

Raptor [41]. Raptor is a simulator for survivability research developed at UVA. It scales to support tens of thousands of nodes. Raptor provides a model for injecting controlled and catastrophic failures into large networks that will be useful for our experiments involving hostile environments.

RoboCup Soccer Server [45]. The Soccer Server provides a semi-realistic physical simulation of robots on a soccer field. It includes models of unreliable physical movement, communication and vision. We have used the Soccer Server to experiment with behavioral swarm primitives in a more hostile and realistic environment than the Santa Fe Swarm simulator [47].

SD/Fast [82]. SD/Fast is a commercial rigid-body simulation system. This software system permits the development of accurate physical simulations of devices and the environments in which they exist. We anticipate using this software to accurately model the scattering of devices on uneven terrain and to simulate the movements of mobile devices.

NS2 [20]: This is the most widely used network simulator today for IP networks. It also has extensive support for wireless simulation that may be used for simulating large sets of devices in a physical environment. SensorSim is an NS2-based simulator that focuses on sensor networks.

Centurion. We also have access to Centurion, a 300 machine cluster developed at UVA [46]. Centurion has been used successfully in several large scale simulation experiments. We will be able to use Centurion to run massive scale simulation experiments as necessary to develop.

4. Impact Summary

Our goal is to develop a new program paradigm for the environment-aware, massively distributed applications of the coming decades. While it would be unreasonable to expect to meet the many challenges towards that goal within three years, we do expect to make concrete progress on several fronts. In particular, we expect to have:

- Designed the Seurat programming language for swarm computing, and described its semantics in a precise way.
- Developed a library of communication, organization and behavioral primitives. For each primitive, we will have a functional specification, and multiple implementations with different non-functional properties.
- Developed theories for combining implementations of primitives. Identified and defined a class of implementations which can be combined with predictable properties.
- Gained an understanding of the scaling properties of swarm programs. Developed tools and methods for managing massively scalable applications.

We request 3 years of funding. We anticipate achieving these milestones by the end of each year:

Year 1:

- Designed the Seurat programming language and languages for specifying nonfunctional requirements and device and environment properties.
- Produced results from simulating combinations of primitives in different environments.
- Developed preliminary theoretical foundations and tools for analyzing aggregate behavior of a large number of devices.
- Analyzed the critical mass for a number of primitive behaviors. Validated these results with simulation experiments.
- Invented techniques for casting non-functional assurance problems into feedback performance control problems.

Year 2:

- Developed a prototype swarm program synthesizer that takes a Seurat program, nonfunctional requirements description, primitives library, device and environment models and produces individual device programs.
- Produced formal descriptions of the semantics of our programming and specification languages.
- Investigated the composition properties of classes of implementations of primitive behaviors and developed a theory for composition that addresses scale and nonfunctional properties.
- Expanded the primitives library to include adaptive algorithms that trade-off resource consumption for performance.
- Developed an experimental physical test bed of static motes with sensors and radio transceivers.
- Implemented distributed protocols for aggregate behavior control based on theoretical results from year 1 in both simulated and physical experiments.
- Implemented a communication micro-protocol stack for small devices including link, network, and transport layer functionality that addresses the concerns of section (Implementing the primitives).

Year 3:

- Produced a complete swarm program synthesizer that is capable of generating device programs for a large (well-defined) class of applications and a range of number and capabilities of devices and environments.
- Produced a library of swarm primitives that is sufficiently comprehensive to be able to produce swarm programs with a wide range of nonfunctional properties.
- Produced analytical and experimental results on the composition of primitive implementations. Specified a set of implementation properties that lead to predictable composition properties.
- Extended the physical testbed to include mobile devices that can alter the environment. Conducted experiments using the physical testbed with applications that store and manipulate state in the environment.

We will disseminate our results through publications, talks, web sites, course materials and software. The proposers have successful track records for distributing and supporting software[3] and believe it is worth the significant effort require to make software available and useful to other researchers both to validate our own work and to foster a vibrant research community.

5. Results of Prior NSF Support

Tarek Abdelzaher is currently funded by an NSF CAREER award ("A Pull-Based Architecture for Active Web Content Replication", CCR-0093144), NSF grant ANI-0105873 ("A Framework for Utilization-Based Absolute Delay Guarantees Using Adaptive Data Prefetching") and is a co-PI on NSF grant CCR-0098269 ("A Control-Theoretic Approach to Performance Guarantees in Performance-Critical Systems"). The first two grants develop different distributed algorithms for caching and content management in a web environment. The second develops performance feedback control algorithms for software systems. These grants resulted in multiple publications [26,27,28,29,30,32,33,34] and software prototypes that provide a good analytic foundation for software control as well as expertise that may be leveraged in a deeply distributed system context with performance assurances.

David Brogan is co-investigator on a recent NSF CISE Research Resources award (0130800) that funds the purchase of the hardware required to build an immersive display system. This hardware will be used to support Brogan's research of simulation level of detail as a perceptually acceptable simplification technique for computer animation.

David Evans has been funded by NSF CAREER ("Programming the Swarm", CCR-0092945) since March 2001. The proposed work complements well both the research and educational work of the CAREER proposal. The work funded under NSF CCR-0092945 has contributed to five publications [47, 50, 56, 57, 58] and two papers currently in submission. The grant funds one graduate student. She is focusing on security properties of stigmeric algorithms, in particular their resistance and vulnerability to certain classes of attack. Two undergraduates have been supported since July 2001 using an REU supplement (GA10183). Ten undergraduate students are currently working (as volunteers, independent study projects, and on undergraduate thesis projects) with the PI on research directly connected to this project.

[3] David Evans has distributed and supported LCLint, an extensible tool for lightweight static analysis that exploits program annotations to detect likely security vulnerabilities, memory management problems, and other common errors in C programs since 1994 [47, 49, 50]. LCLint has several thousand active users and is described in several articles and books directed at working programmers [51, 52, 53, 54, 55].

References

1. Lili Qiu, Venkata Padmanaban, Geoffrey M Voelker. *On the Placement of Web Server Replicas*. Proc. IEEE INFOCOMM 2001.
2. Chalermek Intanagonwiwat, Ramesh Govindan and Deborah Estrin. *Directed Diffusion: A Scalable and Robust Communication Paradigm for Sensor Networks*. In Proceedings of the Sixth Annual International Conference on Mobile Computing and Networks (MobiCom 2000), August 2000, Boston, Massachusetts.
3. John Heidemann, Fabio Silva, Chalermek Intanagonwiwat, Ramesh Govindan, Deborah Estrin, and Deepak Ganesan. *Building Efficient Wireless Sensor Networks with Low-Level Naming*. In Proceedings of the Symposium on Operating Systems Principles (SOSP 2001), Lake Louise, Banff, Canada, ACM. October 2001.
4. Ya Xu, John Heidemann, and Deborah Estrin. *Geography-informed Energy Conservation for Ad Hoc Routing*. Proceedings of the Seventh Annual ACM/IEEE International Conference on Mobile Computing and Networking (MobiCom 2001), Rome, Italy, July 16-21, 2001.
5. N. Bulusu, J. Heidemann and D. Estrin. *GPS-less Low Cost Outdoor Localization For Very Small Devices*. IEEE Personal Communications, Special Issue on Smart Spaces and Environments, Vol. 7, No. 5, pp. 28-34, October 2000.
6. Radhika Nagpal. *Organizing a Global Coordinate System from Local Information on an Amorphous Computer*. MIT AI Memo 1666, August 1999
7. Young-Bae Ko and Nitin H. Vaidya. *Location-Aided Routing(LAR) in Mobile Ad Hoc Networks*. In Proceedings of the Fourth Annual ACM/IEEE International Conference on Mobile Computing and Networking (MobiCom 1998), ACM, Dallas, TX, October 1998.
8. C. E. Perkins and E. M. Royer. *Ad-hoc On Demand Distance Vector Routing*. 2nd IEEE Workshop on Mobile Computing Systems and Applications (WMCSA'99), New Orleans, Louisiana, February 1999.
9. Charles E. Perkins and Pravin Bhagwat. *Highly dynamic Destination-Sequenced Distance-Vector routing (DSDV) for mobile computers*. In SIGCOMM Symposium on Communications Architectures and Protocols, (London, UK), pp. 212-225, Sept. 1994.
10. M. Charikar, S. Guha, E. Tardos and D.B. Shmoys. *A constant factor approximation algorithm for the k-median problem*. Proceedings of the 31st Annual ACM symposium on Theory of Computing.
11. M. Charikar and S.Guha. *Improved Combinatorial Algorithms for the Facility Location and K-median Problems*. In Proc. Of the 40th Annual IEEE Conference on Foundations of Computer Science, 1999.
12. N. Bulusu, J. Heidemann and D. Estrin. *Adaptive Beacon Placement*. Proceedings of the 21st International Conference on Distributed Computing Systems (ICDCS-21), Phoenix, Arizona, April 2001.
13. John Heidemann, Fabio Silva, Chalermek Intanagonwiwat, Ramesh Govindan, Deborah Estrin, and Deepak Ganesan. *Building Efficient Wireless Sensor Networks with Low-Level Naming*. In Proceedings of the Symposium on Operating Systems Principles (SOSP 2001), Lake Louise, Banff, Canada, ACM. October 2001.
14. Tomasz Imielinski and Samir Goel. *DataSpace - querying and monitoring deeply networked collections in physical space*. IEEE Personal Communications Magazine, Special Issue on Networking the Physical World, October 2000.
15. Philippe Bonnet, J. E. Gehrke, and Praveen Seshadri. *Querying the Physical World*. IEEE Personal Communications, Vol. 7, No. 5, October 2000, pages 10-15. Special Issue on Smart Spaces and Environments.

16. J. Hill, R. Szewczyk, A. Woo, S. Hollar, D. Culler, and K. Pister. *System architecture directions for network sensors*. ASPLOS 2000.
17. *Development platform for self-organizing wireless sensor networks*. Proc. SPIE, Unattended Ground Sensor Technologies and Applications, Vol. 3713, p. 257-268.
18. A Compendium of NP optimization problems. http://www.nada.kth.se/viggo/problrmlist/compendium.html
19. *Self-organizing distributed sensor networks*. Proc. SPIE, Unattended Ground Sensor Technologies and Applications Vol. 3713, p. 229-237.
20. The ns-2 simulator. http://www.isi.edu/nsnam.
21. *Distributed Operating Systems*. Andrew S. Tanenbaum Prentice Hall.
22. Guillaume Pierre, Maarten van Steen, andAndrew S. Tanenbaum. *Self-replicating Web document*s, Technical Report IR-486, Vrije Universiteit, Amsterdam, February 2001,
23. The Case for Geographical Push Caching. Proceedings of the Fifth Annual Workshop on Hot Operating Systems, Orcas Island, WA, May 1995
24. E. M. Royer and C. E. Perkins, Multicast operation of the ad-hoc on-demand distance vector routing protocol, in Proc. of ACM/IEEE Intl. Conference on Mobile Computing and Networking (MOBICOM), Aug. 1999
25. A Survey of Multicast Technologies (2000) Vincent Roca, Luís Costa, Rolland Vida, Anca Dracinschi, Serge Fdida September 2000.
26. Kevin Skadron, Tarek Abdelzaher, and Mircea Stan, ``Control-Theoretic Techniques and Thermal RC Modeling for Accurate and Localized Dynamic Thermal Mangement," International Symposium on High Performance Computer Architecture, Cambridge, MA, Feb 2002.
27. Seejo Sebastine, Kyoung-Don Kang, Tarek F. Abdelzaher, Sang H. Son, ``A Scalable Web-Based Real-Time Information Distribution Service for Industrial Applications," The 27th Annual Conference of the IEEE Industrial Electronics Society, Denver, Colorado, December 2001.
28. John Stankovic, Tian He, Tarek Abdelzaher, Mike Marley, Gang Tao, Sang Son, ``Feedback Control Scheduling in Distributed Systems," IEEE Real-Time Systems Symposium, London, UK, December 2001.
29. Tarek Abdelzaher, Chenyang Lu, ``Schedulability Analysis and Utilization Bounds for Highly Scalable Real-Time Services," IEEE Real-Time Technology and Applications Symposium, TaiPei, Taiwan, June 2001.
30. Chenyang Lu, Tarek Abdelzaher, Jack Stankovic, Sang Son, ``A Feedback Control Approach for Guaranteeing Relative Delays in Web Servers," IEEE Real-Time Technology and Applications Symposium, TaiPei, Taiwan, June 2001.
31. Tarek Abdelzaher, Kang G. Shin, Nina Bhatti, ``Performance Guarantees for Web Server End-Systems: A Control-Theoretical Approach," Accepted to IEEE Transactions on Parallel and Distributed Systems , 2001.
32. Ying Lu, Avneesh Saxena, and Tarek F. Abdelzaher, ``Differentiated Caching Services; A Control-Theoretical Approach," International Conference on Distributed Computing Systems, Phoenix, Arizona, April 2001.
33. Tarek F. Abdelzaher and Chenyang Lu, ``Modeling and Performance Control of Internet Servers," Invited Paper, 39th IEEE Conference on Decision and Control, Sydney, Australia, December 2000.
34. Chenyang Lu, John A. Stankovic, Tarek F. Abdelzaher, Gang Tao, Sang H. Son and Michael Marley, ``Performance Specifications and Metrics for Adaptive Real-Time Systems," IEEE Real-Time Systems Symposium, Orlando, Florida, December 2000.
35. Tarek F. Abdelzaher, ``An Automoated Profiling Subsystem for QoS-Aware Services," IEEE Real-Time Technology and Applications Symposium, Washington D.C., June 2000.

36. Tarek F. Abdelzaher and Kang G. Shin, ``QoS Provisioning with qContracts in Web and Multimedia Servers," IEEE Real-Time Systems Symposium, Pheonix, Arizona, December 1999.
37. Tarek F. Abdelzaher, Nina Bhatti, ``Adaptive Content Delivery for Web Server QoS," International Workshop on Quality of Service, London, UK, June 1999.
38. Tarek F. Abdelzaher, Nina Bhatti, ``Web Content Adaptation to Improve Server Overload Behavior," International World Wide Web Conference, Toronto, Canada, May 1999.
39. Tarek F. Abdelzaher, Kang G. Shin, ``End-host Architecture for QoS-Adaptive Communication," IEEE Real-Time Technology and Applications Symposium, Denver, Colorado, June 1998.
40. Tarek F. Abdelzaher, ``QoS Adaptation in Real-Time Systems," Ph.D. Thesis, University of Michigan, August 1999.
41. John Knight, Robert Schutt, Kevin Sullivan, "A System for Experimental Research in Distributed Survivability Architectures". UVA Technical Report CS-2000-29. August 2000.
42. William Oliver. *Analyzing Group Behavior: Developing a Tool to Evaluate Swarm Programs*. UVA Senior Thesis in progress, expected completion May 2002.
43. Swarm Development Group. www.swarm.org.
44. Ryan Persaud. *Investigating the Fundamentals of Swarm Computing*. UVA Senior Thesis. March 2001
45. Itsuki Noda. *Soccer server: a simulator of RoboCup*. JSAI AI-Symposium, 1995.
46. Legion Group. Centurion Applications Page. http://legion.virginia.edu/centurion/Applications.html
47. Keen Browne, Jon McCune, Adam Trost, David Evans and David Brogan. *Behavior Combination and Swarm Programming: University of Virginia Team Description*. To appear in RoboCup International Symposium 2001, Lecture Notes in Artificial Intelligence. Springer-Verlag 2002.
48. David Evans, John Guttag, Jim Horning and Yang Meng Tan. *LCLint: A Tool for Using Specifications to Check Code*. Proceedings of the SIGSOFT Symposium on the Foundations of Software Engineering. p. 87-96. December 1994.
49. David Evans. *Static Detection of Dynamic Memory Errors*. Proceedings of the SIGPLAN Conference on Programming Language Design and Implementation (PLDI '96). Philadelphia, PA. p. 44-53. May 1996.
50. David Evans and David Larochelle. *Improving Security Using Extensible Lightweight Static Analysis*. To appear in IEEE Software, Jan/Feb 2002.
51. David Santo Orcero. The Code Analyser LCLint. In Linux Journal, May 2000.
52. Pramode C E and Gopakumar C E. Static checking of C programs with LCLint. In Linux Gazette, March 2000.
53. David Hanson. *C Interfaces and Implementations: Techniques for Creating Reusable Software*. Addison Wesley Professional Computing Series, 1996.
54. Richard Heathfield (Editor), Lawrence Kirby (Editor). *C Unleashed*. Sams. 2000.
55. Michael Welschenbach. *Cryptography in C and C++*. APress. May 2001.
56. Joel Winstead and David Evans. *Structured Exception Semantics for Concurrent Loops*. Fourth Workshop on Parallel/High-Performance Object-Oriented Scientific Computing. October 2001.
57. Chenxi Wang, Antonio Carzaniga, David Evans and Alexander Wolf. *Security Issues and Requirements for Internet-Scale Publish-Subscribe Systems*. To appear in Hawaii International Conference on System Sciences, January 2002.
58. David Larochelle and David Evans. *Statically Detecting Likely Buffer Overflow Vulnerabilities*. 10th USENIX Security Symposium, August 2001.
59. David Rotman. *Molecular Computing*. Technology Review. May/June 2000.

60. J. M. Kahn, R. H. Katz and K. S. J. Pister. *Next Century Challenges: Mobile Networking for Smart Dust*. ACM/IEEE International Conference on Mobile Computing and Networking (MobiCom 99). p. 217-278. August 1999.
61. IEEE Std 802.11-1997: Short description of the standard. http://www.manta.ieee.org/groups/802/11/main.html
62. *Bluetooth Forum*. http://www.bluetooth.net.
63. David Gelernter. *Generative communication in Linda*. ACM Transactions on Programing Languages and Systems. Volume 7, Number 1, p. 80-112. January 1985.
64. Nagpal, Organizing a Global Coordinate System from Local Information on an Amorphous Computer, MIT AI Memo No. 1666, August, 1999
65. S. Goss, S. Aron, J.L. Deneubourg, and J.M. Pasteels. Self-organized shortcuts in the Argentine ant. *Naturwissenschaften* **76**, 579-581, 1989.
66. R. Beckers, J.L. Deneubourg, and S. Goss. Trails and U-turns in the selection of the shortest path by the ant Lasius niger. *Journal of Theoretical Biology*, **159**, 397-415, 1992.
67. T. D. Seeley, C. A. Tovey, and J. H. Vande Vate. The pattern and effectiveness of forager allocation among food sources in honey bee colonies. *Journal of Theoretical Biology* 160:23-40, 1993.
68. Thomas D. Seeley. *Wisdom of the Hive*. Harvard University Press, 1996.
69. S. Veherencamp, Handbook of Behavior Neruobiology, Volume 3: Social Behavior and Communication.
70. J. M. Cullen and E. Shaw and H. A. Baldwin, Methods for Measuring the Three-dimensional Structure of Fish Schools, Animal Behavior, Volume 13, 534-543
71. R.C. Arkin. Dimensions of communication and social organization in multi-agent robotic systems. In *Proceedings of the Second International Conference on Simulation of Adaptive Behavior: From Animals to Animats 2*, pages 486-493, 1992.
72. M. S. Fontan and M.J. Mataric. Territorial multi-robot task division, *IEEE Transactions on Robotics and Automation,* 14(5), Oct 1998, pages 815-822.
73. P. Stone and M. Veloso. Team-partitions, opaque-transition reinforcement learning. Technical report, Carnegie Mellon University, April 1998.
74. C.W. Reynolds. Flocks, herds, and schools: A distributed behavioral model. In *Proceedings of Siggraph '87*, pages 25-34. ACM SIGGRAPH, July 1987.
75. B.M. Blumberg and T.A. Galyean. Multi-level direction of autonomous creatures for real-time virtual environments. In *Proceedings of Siggraph 95*, pages 47-54. ACM SIGGRAPH, August 1995.
76. X. Tu and D. Terzopoulos. Artificial fishes: Physics, locomotion, perception, behavior. In *Proceedings of Siggraph '94*, pages 43-50. ACM SIGGRAPH, July 1994.
77. S.R. Musse and D. Thalmann. A model of human crowd behavior. In Computer Animation and Simulation '97, Eurographics Workshop, Budapest, pages 39-51. Springer-Verlag, 1996.
78. K. Sugihara and I. Suzuki. Distributed Motion Coordination of Multiple Mobile Robots, Proceedings of the 1990 IEEE International Conference on Robotics and Automation, 138-143. 1990.
79. D. Brogan and J. Hodgins. Group behaviors for systems with significant dynamics. *Autonomous Robots*, 4:136-153, 1997.
80. D. Brogan, R. Metoyer, and J. Hodgins. Dynamically simulated characters for virtual environments. IEEE Computer Graphics and Applications, 18(5):58-69, September/October 1998.
81. D. Brogan and J. Hodgins. Group behaviors for systems with significant dynamics. *Proceedings of the 1995 IEEE/RSJ International Conference on Intelligent Robots and Systems*, Vol. 3, pp 528-534, 1995.
82. J. Hodgins, W. Wooten, D. Brogan, and J. O'Brien. Animating human athletics. *Proceedings of Siggraph '95*. In *Computer Graphics*, pp 71-78, 1995.

83. M. Hollars, D. Rosenthal and M. Sherman. SD/Fast, Symbolic Dynamics, 1991.
84. Eric Bonabeau and Guy Théraulaz. *Swarm Smarts*. Scientific American. p. 72-79. March 2000.
85. Camazine, S. et al. *House Hunting by honey bee swarms.* Insectes Sociaux Volume 46 Number 99. p. 348-360. January 1999.
86. Gilbert Waldbauer. *Millions of Monarchs, Bunches of Beetles: How Bugs Find Strength in Numbers*. Harvard University Press. 2000.
87. Paul V. Mockapetris. *Domain Names – Concepts and Facilities*. Network Working Group, RFC 1034. November 1987.
88. William Adjie-Winoto, Elliot Schwartz, Hari Balakrishnan, and Jeremy Lilley. *The design and implementation of an intentional naming system*. Proceedings of the 17th ACM symposium on Operating systems principles. p. 186-201. December 1999.
89. Lakshminarayanan Subramanian and Randy H.Katz. An Architecture for Building Self-Configurable Systems. To appear in IEEE/ACM Workshop on Mobile Ad Hoc Networking and Computing. August 2000.
90. Chalermek Intanagonwiwat, Ramesh Govindan and Deborah Estrin. *Directed Diffusion: A Scalable and Robust Communication Paradigm for Sensor Networks.* To appear in *ACM MobiCom 2000.*
91. Eric Freeman, Susanne Hupfer and Ken Arnold. *JavaSpaces Principles, Patterns and Practice*. Addison Wesley, 1999.
92. Tobin J. Lehman, Stephen W. McLaughry and Peter Wyckoff. *T Spaces: The Next Wave*. IBM Systems Journal, August 1998.
93. Gian Pietro Picco, Amy L. Murphy and Gruia-Catalin Roman. *LIME: Linda Meets Mobility*. Proceedings of the 1999 International Conference on Software Engineering. p. 368-377. May 1999.
94. Daniel Coore. *Botanical Computing: A Developmental Approach to Generating Interconnect Topologies on an Amorphous Computer.* MIT PhD Thesis. December 1998.
95. Harold Abelson, Don Allen, Daniel Coore, Chris Hanson, George Homsy, Thomas F. Knight, Jr., Radhika Nagpal, Erik Rauch, Gerald Jay Sussman, and Ron Weiss. *Amorphous Computing*. Communications of the ACM, Volume 43, Number 5, p. 74-83. May 2000.