VizION: The VizClass Interface Operating Network

Dirk Groeneveld and Tara Hutchinson and Falko Kuester

November 1, 2005

Abstract

Ubiquitous computing environments are presenting a new set of challenges to application developers. Applications have to be compatible with a wide variety of devices and be resilient in the face of partial system failures. Further, with devices and applications gaining quickly in complexity and size, the software systems have to be scalable, guaranteeing quick responses to user input under all conditions. Since it is still a relatively new field, ubiquitous computing still lacks the kind of programming paradigms that have emerged in many other fields of software engineering. This paper introduces VizION, a middleware layer aimed at hiding many of the intricacies of ubiquitous computing environments from the application developer, providing a simple, unified platform that encourages software development by making it easy. An added goal for the middleware is that of flexibility, since software development for these environments is broad in scope and the middleware should not constrain the developer in any way. The main contributions of the VizION system are in two areas, (1) its underlying structure is modeled loosely after distributed hash tables, avoiding central entities that could become performance bottlenecks or single points of failure, and (2), the unique message system works in the distributed environment to deliver messages efficiently and reliably without concerning the developer with the details of message delivery. VizION was tested and evaluated in VizClass, a ubiquitous computing classroom specifically geared towards digitally enhanced engineering education, and the obtained results are presented.

1 Introduction

1.1 Motivation

With computer technology becoming ever more pervasive, more and more computers are being built transparently into homes, offices, schools, and a broad range of mobile and stationary appliances. As a result, we are moving closer to the vision of ubiquitous computing that Xerox PARC set forth in 1988 [13, 14], a vision that has since been adopted by researchers, educators and practitioners.
around the globe. The new technology based environments require a middleware that allows the control and coordination of many different devices. The middleware has to fulfill a multitude of requirements. It has to tie a potentially large number of devices together into a homogeneous system that is easy to use and encourages developers to build on it and extend it. As a special requirement, it has to be able to interface with existing drivers and other software as seamlessly as possible, not only to avoid repetitious code, but also to communicate with device drivers and applications that are not openly accessible. At the same time, the middleware has to be flexible enough to accommodate all the different devices that might become part of a pervasive computing system, ranging from RFID tags to USB sticks to large scale storage clusters, embedded sensor nodes to mobile phones and PocketPCs to compute and display clusters, or from single projectors to wall-sized projection screens. Part of that flexibility is the requirement of portability. The system should be able to cope with components running on a variety of platforms and operating systems. The challenge is that ubiquitous computing environments tend to be very volatile. Parts might enter or leave the system without warning, some appliances might fail unexpectedly, and they are all connected with something as unreliable as a wireless or wired network. In this environment, the middleware has to ensure the correct operation of all devices that are unaffected by change and keep latencies at a level that goes unnoticed by the end user of the system.

1.2 Related work
Marc Weiser and his team initiated research the field of ubiquitous computing in 1988 at Xerox PARC [13, 14]. Other projects developed this idea further, among them the Interactive Workspaces Project [4] at Stanford, the "Classroom 2000" at the Georgia Institute of Technology [1] and the "Office of the Future" project at the University of North Carolina at Chapel Hill [9], and the VirClass project at the University of California, Irvine [2]. These and other groups developed their own middleware frameworks for their ubiquitous computing environments. Of note are Stanford's IRIS system [7], the GAIA operating system [11] and the PICO system [5].

The different middleware frameworks address the same challenges, but they are targeted towards different applications and use different approaches. IRIS uses a centralized event heap for communication between the different parts of the system [3]. The VisION system, presented in this paper, uses a highly distributed approach that avoids centralized entities completely, following the example of PICO. PICO, however, goes beyond the pervasive computing room and targets large installations the size of entire cities. The GAIA system is also highly distributed and focuses on room-size installations. The main difference between GAIA and VisION is that GAIA implements many features as part of the framework that are applications on top of the framework in VisION.

Interesting ideas about distributed networks may be found by studying peer to peer file sharing networks such as Gnutella [5, 10]. They provide a prime example of efficient coordination of many different nodes on a network, on a
scale larger than in most pervasive computing environments. Principles of the Gnutella network (and others) were generalized into the concept of Distributed Hash Tables (DHTs) [15]. One of the first DHTs, and a representative example for them, is the Chord system [12]. VizION's underlying structure is loosely based on some of the principles found in DHTs.

2 Testbed for VizION: VizClass

The development of VizION emerged from a project at the University of California, Irvine to develop a ubiquitous visualization classroom, termed VizClass [2]. The goal of this project is to build a digital classroom aimed at providing engineering students with a more hands-on experience that current chalkboard-based learning cannot provide (see Figure 1).

![Figure 1: Student working in VizClass on biomedical visualizations in 2D and 3D](image)

At the heart of the room are three rear-projected digital whiteboards mounted adjacent to each other. Each digital whiteboard has a diagonal of 72 inches, providing a combined projection surface of about 1.6 meters in height and 4.4 meters in width. There are many ways this surface can be put to use. One way is to use them as one large unified screen, but more frequently they are used separately. For example, often one board shows a PowerPoint presentation, while an example is provided on another. The digital whiteboards have a touch-sensitive surface that can be used in two possible fashions. One is to use the surface of the board as a replacement for a mouse. Since usually there is no mouse available during a presentation, this is a very practical way to interact with the systems that are connected to the boards. The other mode is to use one of four pens or the eraser that are supported as state devices, allowing at most one of them to be active at any given time. This setup enables users to fuse information from a broad range of sources, including presentation slides, video sequences, animation, simulation results or other visuals, generated by applications running on the connected host computer, with digital ink and user integration. All changes
can be saved for review and reuse at a later time, or transmitted to another local or remote device.

The other main system of the room is the 3D stereo projection screen. Two projectors, one for each eye, rear-project polarized images onto one 1.8 meter by 2.4 meter surface. Polarizing glasses are used to extract the correct image for each eye. Having a 3D system next to the 2D systems offers a wealth of possibilities for engineering education. To name a few, 2D models can be visualized in 3D space, real-world object can be modeled interactively, and three-dimensional concepts can be shown without the need for two-dimensional representations.

Other systems such as cameras, speakers and microphones make the room a complete pervasive computing environment. Allowing users to record and replay entire work sessions, or represent data through sound instead of images.

In this context, VizION has to be powerful enough to meet the demands that VizClass places on it, while remaining flexible enough to support future developments, including ideas that users may not have thought of yet. Most importantly, it must be easy for new users to adopt VizION technology, to encourage continuous use and development.

3 VizION Control Layer

Conventionally during software development, large applications are split into a number of small, reusable modules. VizION supports this notion by providing the concept of a VizION node. A node can be likened to an object in a traditional programming environment, but there are many important differences in their use and their capabilities. Most importantly, they do not have to run on the same physical machine since they communicate over a network (Figure 2). Every node provides ideally one, possibly more services to all other nodes in the system and in return uses services provided by other nodes. An example for a service may be a projector node that provides the service of turning the connected projector on or off, adjusting the contrast, and modifying other projector controls. The programmer can easily query the nodes that exist on the system and use them without caring about their physical location or other details beyond the capabilities these nodes provide. Through a process called "monitoring", which is described in detail below, changes in the list of active nodes can be observed and acted upon. For example, an application that allows the user to turn projectors on or off would monitor the system for nodes with the capability of manipulating projectors, enabling it to react to new projectors joining the system and existing ones leaving.

By splitting up the system into these nodes, we achieve fault tolerance, flexibility and scalability. The system can grow freely, since the computational load is split among the individual components. While more components produce a higher load, they also provide more resources to handle that load. The complete absence of a centralized entity avoids bottlenecks in the system that would become a problem under high loads when many components, i.e. many nodes interact at the same time. It also avoids a single point of failure. Nodes might
fail without warning, but since no single node is essential to the functioning of the entire system, a large part of the system can remain entirely unaffected by these failures. Conversely, if a node is being added, all other parts of the system will continue to run without change unless interacting with the new node is required. Of course, sometimes it cannot be avoided that a failure affects other parts of the system. For example, when a projec"tor node fails, its capabilities are no longer available and if an application depends on these capabilities it cannot function. For these cases, VizION has a defined way of handling errors and passing them back to the application developer.

3.1 Internal Organization of the VizION system

Being decentralized and independent from each other, VizION nodes need a mechanism to discover other components of the system and monitor their status. They achieve this by organizing themselves into a mesh structure in which every node has at least one, but no more than six neighbor nodes (Figure 3). When a new node starts up, it needs the address of at least one other node that is already part of the system. Any node will do and there is no preferential treatment. If the known node has less than six neighbors, the new node becomes its neighbor. Regardless of whether this operation was successful or not, the new node asks for the neighbors of the known node and tries to become their neighbor in the same fashion, until four neighbors have been found. Should the number of neighbors drop below four later, the same process is applied again until four or more neighbors have been found. Once a new node is connected to the system, it broadcasts a "birth message" to the rest of the system.

Unfortunately, in a volatile environment with no central control such as VizION, these processes cannot be trusted to be completely accurate. Nodes might fail during the connecting process, and more than one node at a time might attempt to join the system. Because of these issues it may happen that a node ends up with more than six neighbors. Further, if two nodes are connected to each other but both of them have six connections, they are maintaining a connection that is unnecessary for the mesh properties to hold, wasting resources, and they are preventing new nodes from joining quickly. To avoid this accumulation of useless connections over time, a VizION node checks its number of
neighbors every time a new neighbor joins, and attempts to purge a connection if that number is higher than five. Purging only occurs if no neighbor drops below the mark of four neighbors in the process; otherwise all connections are retained.

Note that the numbers four, five and six in this methods are parameters that can be tuned to suit a given environment. The given numbers were chosen small to ease the study of the behavior of the mesh, however, in a production environment much higher values might be used to increase reliability of the network.

Neighbor nodes stay connected to each other by sending periodic heartbeats. If sending a heartbeat or any other interaction with a node fails, the node is considered dead and a special “death message” is generated and broadcast throughout the system, such that other nodes can remove their stale references to the node that died or take other actions such as updating a graphical display. If multiple nodes detect the death of a node at the same time, two death messages are generated, but a node that has already received one of them does not forward the other.

The major feature of this system is that every node has to store and process only a constant amount of local information. Therefore, the resources required by each node do not increase with the size of the network, allowing for exceedingly large networks. The graph in Figure 4 illustrates this. It shows the total number of network connections in the system as more and more nodes are being added. Every time a node gets added we see a spike of activity as the new node
finds its place in the network. In this test run, ten nodes were added about ten time intervals apart. When the number of nodes in the network is less than five, the activity is much higher as all nodes are actively trying to find neighbors. After that, the activity baseline is determined by the heartbeat rate, whose frequency has been increased to a high level to show the effect. It is apparent that the total activity on the network grows linearly with the number of nodes, indicating that every individual node bears a constant load.

![Graph](image)

Figure 4: Network activity in a small VisION network. Nodes are added one every 10 time intervals, resulting in a spike of activity as the new node configures itself to be part of the network.

### 3.2 Name, Description and the unique ID

Every node carries some static information with it. The simplest forms of this static information are the name and the description of a node. Both have no technical significance, but they identify components to the administrator of the VisION network. While there are no hard limits on the size of both entries, by convention the name is short enough to fit on one line of a graphical user interface, and the description is a sentence or two that gives more information about the function the node performs. Neither the name nor the description can change during the lifetime of a node.

To distinguish nodes reliably and efficiently, every node has a unique ID in the VisION system. The ID is a string with the only requirement that it is unique in space and time. That means that when a node fails and restarts, it will receive a different ID. To make sure that the ID observes the required properties, it is generated automatically during startup as a combination of IP address, IP port, and system time.

### 3.3 Types

In a way, the type system performs the same function as the description field, but it is intended for communication between the nodes. Types are queried
to find out which exact functionality a node exports to the rest of the system. They are organized in a hierarchy, and one node can have multiple types. The easiest way to illustrate this is by example.

A camera node that is driving a Sony camera of type EVI-D100 has the functionality to zoom, pan and tilt a camera, and of course it offers a video stream. In the type system, that would mean that the node could have the type "/device/camera/Sony/EVI-D100" to completely identify the camera that is being used. If another part of the system is interested in all the cameras made by Sony, it will query for nodes of the types "/device/camera/Sony/*". Much more likely of course is the case when all camera nodes are wanted, regardless of the manufacturer. In that case, the search query would be "/device/camera/*". However, nodes can carry more than one type. The camera node, since it also provides a video stream, could carry the type "/output/video/raw", if raw was the encoding of the video being offered. More types are possible, for example the capability to zoom could be "/input/zoom".

The exact names of the types are up to the programmer of the node, but to be useful, they should follow the established conventions. This setup is analogous to a file system which supports arbitrarily named directories and files, but only becomes useful when "/tmp/" or "/usr/bin/" can always be expected to hold temporary files or executables, respectively.

3.4 The Monitoring System - VizION's Discovery Process

Since types are loosely equivalent to the capabilities a node offers, nodes are usually searched for by their type. Two modes of searching exist and together form the so-called "monitoring system". The first mode of operation is the straightforward search, which happens asynchronously. The searching node broadcasts the types it is looking for, and receives positive answers from nodes whose types match the requirements. Every time such a positive answer is received, VizION calls a handler function that was supplied by the programmer. Broadcasts do not scale well and should therefore be kept to a minimum. The broadcast search can be replaced with an overlay that sorts nodes by their types and allows searching in O(log n) time. However, thus far experience has shown that the impact of searches is in fact small during the average operation of a VizION network, since the second mode of operation handles a major part of the searches. It simply waits for birth messages and reports a match when a new node matches the search requirements. Together, these two systems can find an existing node that provides a certain capability, whether it is already part of the system or will join in the future. The programmer is free to work with the two modes individually, but usually it is more convenient to setup a monitor for a given type of node.

3.5 Storing Read-Only Values as Attributes

Every node has a variable number of attributes that provide general information about the type of service offered. These attributes are simply read-only
values that can be queried from other nodes. Values can be integers, strings, or containers such as lists or dictionaries (mappings), which in turn contain other values. The value of an attribute does not change as long as a node is running. Consequently, the interface to these modules is quite simple, as Figure 5 illustrates. Storing read-only values can be very useful. In our example of a camera node, read-only values could be the resolution of the video stream or the physical location of the camera in all three dimensions.

![Figure 5: A schematic of the attribute system implemented in VisION](image)

3.6 Messages: The Main Mode of Communication

Nodes communicate with each other using messages. The message system falls under the category of coordination based systems. Its logic is shown in Figure 6. Messages are sent directly from one node to another without a central message dispatcher, and consist of a number of components.

![Figure 6: A schematic of the message system implemented in VisION](image)

Every message has a name, which other nodes can use to identify it, and use to subscribe to it. When a remote node subscribes to a message on the local node, the local node adds the remote node to the list of recipients for that message and will subsequently send the message to the remote node whenever it is generated. Canceling a subscription is fairly straightforward, only requiring the remote node to send an unsubscribe command to the local node, which then removes the remote node from the list of recipients.

Messages also carry labels, analogous to the types the nodes carry. Just like the node types, the message labels are organized in a hierarchical fashion, and
one message can have more than one label. All messages with a given name have a set of standard labels, and each individual message can have additional labels. The two categories are collated into a label collection. The principle is illustrated in Figure 7, where the blue labels are the standard labels, and the green labels are added when the message is generated. This particular message is sent out by a USB login node when user “marvin” from the group “users” logs in. Since the USB node always uses USB for authentication, it is a standard label rather than an additional one, whereas the name of the user and his group might change, and therefore is added when the message is sent.

![Figure 7: Anatomy of the message labels of a login message sent by a USB login node](image)

Subscriptions to a message can be made conditional on these labels. A condition can be any combination of boolean logic operators. In this context, a predicate is true if a label (which is given in the subscription) matches the collection, and false if it does not. We say that a label matches the collection when it is contained in it. Note that the labels that a message carries do not have to be sent over the network, and as a result, the label scheme for the messages can be arbitrarily rich and expressive without hindering performance.

Finally, messages contain arguments, which can be simply passed along with the message. When a message is received by a node, it triggers the execution of a message handler that reads these arguments and acts accordingly. The message handler also receives the name of the message and the labels it was sent with, allowing the programmer of the message handler to use the same handler function for different messages.

Many of these concepts and the reasoning behind them become apparent during testing. Consider the case of a login node that handles all sorts of login and authentication methods such as password entry, USB authentication or RFID identification. Whenever a user logs in, this node sends out a message informing other interested nodes that a login has occurred. The arguments to such a message could be the user name, the user groups the user belongs to or which authentication method was used. The message would also carry labels that identify exactly the type of message. For example, if the user logged in using RFID as her authentication method, one label would be “/login/method/RFID”. A node that monitors security related information might listen for those messages, but it only wants to receive the message if the user was authenticated by RFID. It would then make its subscription to the message conditional on the fact that “/login/method/RFID” is part of the collection of labels of the message. Note
that this is just an example, the actual login procedure is more involved.

3.7 Commands: A RPC wrapper

Often, the message-based approach of communicating is insufficient or counterintuitive to the programmer for a given task. Consider the example of a sound mixer node that offers the functionality of setting the volume of a speaker to a certain level. To achieve this with messages, the mixer node would have to subscribe to the changeVolume messages of all nodes in the system. Generally, this will be impractical and hurting performance.

Instead, VizION implements commands as an RPC wrapper. During initialization, any number of commands can be specified and a handler function can be attached to them. When the command gets called from another node, this handler function is invoked. On request, every node returns a list of valid commands to other nodes. Our mixer node for example would support a changeVolume command to set the volume level of the speaker.

3.8 Streams: A Flexible Mechanism to Add Extra Functionality

To increase flexibility, nodes can advertise zero or more streams. A client node querying for streams will get a list of stream names with information about how to access them. A stream is simply another part of the node listening for and answering connections on a different network port. The name stems from the fact that streams were added to the nodes as a means to offer video streaming for camera nodes, however, the concept quickly outgrew its original purpose. VizION places no restraints on the implementation or the protocol of a stream, it can even be written in a different language and run as a different process. One major use of that is to make an existing application part of the system by referring to its open network port as a stream. That makes streams flexible while keeping them a part of the VizION system. The simplicity of this concept is reflected in Figure 8.

![Figure 8: A schematic of the stream system implemented in VizION](image-url)
3.9 Design and Implementation

A major design decision was the selection of a stateless protocol for all communication between nodes. After a TCP connection has been established, one node transmits a request. Then the other node processes it and transmits the response, and the connection is terminated. Experiences with a stateful protocol during early versions of VizION have shown that the simplicity that a stateless protocol offers outweighs the extra functionality of other approaches. Consistency is ensured by designing VizION requests such that they perform all necessary functions at once. Another technique is to check for the correctness of the prerequisites for a command, a procedure that has to be done anyways for security reasons, in combination with a consistent mechanism to transmit failures over the network back to the caller of the command. Pervasively following these design guidelines led to interesting results during debugging when the system was able to function in the face of programming errors as severe as deadlocks, which are broken by a timeout.

To reach the goal of compatibility with many different devices, VizION is written in Python [8]. Python runs stably on all major operating systems; Windows, Linux and MacOS. Being an interpreted, high-level language, it offers rapid development and greater safety at an acceptable loss of performance. Its tight connection to C allows developers to reimplement time critical portions of the system in C, and many packages that facilitate seamless interaction with other programming languages already exist.

4 Conclusions

This paper presents VizION, a software framework that introduces new paradigms for software development in ubiquitous computing environments. We describe VizION in the context of a testbed project termed VizClass. In order to do this, we formulate the middleware using independent nodes that communicate with each other through a standardized system of messages. Consisting of these distributed nodes, VizION achieves the stated goals of fault tolerance and scalability while making it easy to build new applications on the platform that it provides.

Acknowledgments

This research is supported by the National Science Foundation, under Grant Number EIA-0203528. Support for the hardware infrastructure was provided by the California Institute for Information and Telecommunications (CaIT2), the HIP Technology for Teaching Grant under award number 15923 and the Henry Samueli School of Engineering at UC Irvine. In addition, the first author was partially supported by the Summer Undergraduate Research Program (SURP) and the Summer Undergraduate Research Fellowship in Information Technology (SURF-TI) at UC Irvine. The above support is greatly appreciated.
References


