A System Architecture for Ambient Intelligent Environments

Christopher R. Baker

Electrical Engineering and Computer Sciences
University of California at Berkeley

Technical Report No. UCB/EECS-2006-178
http://www.eecs.berkeley.edu/Pubs/TechRpts/2006/EECS-2006-178.html

December 15, 2006

A System Architecture for Ambient Intelligent Environments

by Christopher R. Baker

Research Project

Submitted to the Department of Electrical Engineering and Computer Sciences, University of California at Berkeley, in partial satisfaction of the requirements for the degree of **Master of Science, Plan II**.

Approval for the Report and Comprehensive Examination:

Committee:

Jan M. Rabaey
Research Advisor

(Date)

* * * * * * *

John Wawrzynek
Second Reader

(Date)

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Falling electronics prices have brought sophisticated consumer electronics within the reach of average users. Further, due to the proliferation of different device types, communication, and media coding standards, interoperability between devices will become increasingly important. Advances in the electronics industry have also driven growth in several related areas, in addition to home entertainment, including surveillance, home automation products (*e.g.* lighting, heating, and power), home appliances (*e.g.* laundry machines, fridges, etc.), and ad-hoc wireless sensor networks (AWSN), which promise to add a truly ambient intelligent component to the home. The future home clearly represents an opportunity for the convergence of these different technologies far beyond the level of integration seen today. In fact, the future smart-home will contain a collection of heterogeneous networked devices, capable of distributed computation, dynamic reconfiguration, high-performance media dissemination, and user/environment awareness.

1.1 Ambient Intelligent Environment

We define an Ambient Intelligent Environment (AIE) as an environment in which a network of distributed devices seamlessly cooperate to simplify daily tasks.

We use the term *ambient* to convey a sense of ubiquitous or pervasive computing. Something Mark Weiser called "invisible computing" [27]. Small electronic devices are all around us sensing, recording, and transmitting data. All of these devices are unobtrusively integrated into the surroundings. We see *intelligence* playing an important part of our definition. It describes how the environment assists in daily tasks. For the most part these actions will be taken on behalf of the user given user-specified constraints or prefer-

ences, however, eventually a system may be programed with an ability to learn such things. Finally, we picture a set of distinct (*e.g.* home, office, car), but continuous (*i.e.* seamless transition between them), *environments* in which these devices and users exist and interact. These environments by nature are constantly changing as users and devices come and go, or users make explicit or implicit requests. The environment must support a rich set of devices, must scale appropriately, and offer multiple interaction modalities.

Once an AIE has been developed and deployed, many interesting application classes emerge, particularly for the home and office. Health care (monitoring of body function) for elderly and sick may allow people to live longer, healthier, and more independent lives. Building monitoring and automation will enable further improvements in energy efficiency, where real-time energy prices are used to determine energy usage (using appliances at cheapest energy points) to reduce overall load on the electrical grid. Integrated security systems will provide high-quality, correlated real-time views (motion, sound, video) of every aspect of the environment to improve the safety of private homes and public spaces. Multi-media entertainment, games, and toys present a significant commercial opportunity, where home electronics and online services are integrated to provide instant information, highest-quality viewing and sound, and networked interactive entertainment. For both businesses and homes, audio/video communication will be enabled by technology which makes face-to-face communication a reality, even though that person or group may be on the other side of the planet. Of course these are only some of the applications of this technology. What makes these applications even more appealing is their integration with user identity (personalization) and location, as well as their interaction with each other.

To help realize the exciting potential of an Ambient Intelligent Environment, this report will focus on the necessary systems support (operating systems, networking, etc.). In general, the system architecture for AIEs has the following bottom-up structure: There are a set of devices which are distributed about the home (cell-phone, cameras, sensors, video screens, speakers), interfaced through a hardware abstraction layer (common execution kernel or interface). This interface provides uniform connectivity (*e.g.* using existing Internet protocols) and control. On top of this layer there exist a set of system services for discovery, localization, process management, I/O, etc. Finally, applications are developed to utilize these services. We use a similar structure, however, we propose and abide by a set of properties, known as ZUMA, which affect all layers of the system architecture.

1.1.1 ZUMA

In the past, electronic equipment was sold as vertically integrated boxes. Today, new solutions and standards are introduced faster than the lifetime of many pieces of the home infrastructure. As a result, consumers incrementally upgrade their existing systems. In addition, new products are introduced in the sensor network, security, and home automation spaces, and must be added to the existing system. Thus, it is desirable to design a system that can seamlessly integrate new devices *without* user intervention. The solution outlined in this report supports automatic integration of new devices and efficient use of the functionality they provide. We call this ***Z**ero-configuration*[1].

As alluded to earlier, the interoperability of devices is complicated by the disparity at several levels of the networking stack: different wireless technologies (*e.g.* sensor motes vs. 802.11), device communication protocols, and presentation standards (*e.g.* video encoding). The agreement of a common set of standards is unlikely; moreover, the diverse needs of devices in terms of bandwidth, latency, and power consumption preclude a single set of standards from being successful. Therefore, a method must exist that allows the connection of any set of devices to any other set of devices. The role of the AIE is to provide the necessary levels of indirection to realize this goal. We call this ***U**niversality* and our solution addresses this problem.

Another issue occurs when different users share devices in the home. These users often have different preferences and modes of usage. Computers have dealt with the personalization issue by allowing the creation of separate accounts. However, it is not currently possible to personalize settings for an entire space or home entertainment application (*e.g.* light levels, speaker volume, etc.) and resolve conflicts between users. The system should recognize and accommodate different users according to their properties, permissions, and preferences.

Furthermore, the explosion of different user interfaces presents a nightmare to those wishing to use the current devices in the home. Consider a modern home that has a multimedia center with a television set, DVR, DVD player, a sound system, lighting control and a security system. In the worst case, the user requires a remote control for each device. It is inconvenient and often confusing. The system should enable applications to provide a uniform manner of control for simple manipulation of the home environment. The presented solution will address the issues of enabling personalization and easy manipulation of the environment. We refer to this as ***M**ulti-user optimality*.

[1] We use this term differently than the IETF Zeroconf Working Group and their efforts for automatic simple-network IP configuration.

Current home environments are relatively static, meaning their operation requires manual intervention. It has long been a goal of researchers to make the devices and applications dynamically adapt to changes in the environment. One would like adaptation to the presences of humans, their minute-to-minute desires (*i.e.* both implicit or explicit), and the current environment conditions (*e.g.* a new device arrives). The solution this report discusses will enable such dynamic run-time reconfiguration which we call ***A**daptability* of the system.

Given these four tenets, we have developed a system architecture which embraces these tenets as guiding principles. Important issues such as digital rights management, user privacy, and network security are beyond the scope of this report. We do not preclude the existence of these technologies in our system, however, solutions are either under development by various industry/research groups or are not appropriate for academic research.

1.2 Scenario

We present a scenario that illustrates a subset of the desired behavior of ZUMA for an AIE. A multi-media application was selected for this scenario because with currently deployed technology, it allows the most concrete use case. In this work, the following scenario will serve as a vehicle for discussion and for differentiating ZUMA from other proposed solutions:

> *John walks into a room and turns on the television, which starts a multi-media viewing application. John likes to watch violent sports, and according to his* preferences*, the TV automatically switches to WWE wrestling. The WWE wrestling channel comes in from the set-top box. John watches for awhile, but then his son Adam walks in with a laptop. Adam is a minor, and is not allowed to watch WWE. The television immediately stops showing WWE and puts up a menu to select different content.*
>
> *In the background, the system continues recording the WWE channel and stores the time at which John stopped watching. This allows John to resume watching WWE at a later time.*
>
> *John now selects "Finding Nemo" (from Adam's laptop) to watch with Adam. "Finding Nemo" starts playing on the television.*

This scenario demonstrates aforementioned tenets. *Universality*: the television, a set-top box, and a laptop inter-operating effortlessly together. *Adaptability*: the system detects

non-electronic events (such as the movement of people). This "ambient" functionality is provided by sensor networks. Additionally, content is decoupled from the rendering devices. In order to render content on different devices, it may need to be reformatted and/or trans-coded. *Zero-configuration*: as new devices (*e.g.* the laptop) enter an environment, they are seamlessly integrated. *Multi-user optimality*: The system abides by the preferences and permissions of scenario's users.

1.3 Contributions

This is a very large research area which covers many disciplines. Unfortunately, as a researcher, we have to choose several of these research challenges where we feel we can make an impact. Specifically, the contributions of this work toward realizing an Ambient Intelligent Environment are the following:

- A set of abstractions, specifically for those elements affecting people, content, and system capabilities.
- A system architecture to enable configuration and organization of content and networked heterogeneous devices in a smart-home environment.
- A working prototype demonstrating several of the system architecture concepts.
- Performance results of this prototype.

1.4 Report Organization

This report is organized as follows: Chapter 2 will discuss related research efforts in the areas most relevant to this work: system architecture, resource mapping, and device interoperability. The system architecture proposed in this report will be discussed in Chapter 3. It will include a discussion of the main abstraction, how we envision the applications to operate, and how the system performs run-time refinement and mapping of these applications. Along the way, different characteristics of the architecture will be pointed out as satisfying requirements for ZUMA. Chapter 4 will detail portions of the system architecture which were implemented in our prototype. Specifically, it will discuss the design and implementation of the Home Gateway Demo. Chapter 5 will discuss the evaluation and results of those features implemented. Finally, opportunities for future research will be given in Chapter 6, and conclusions given in Chapter 7.

Chapter 2

Background

This chapter will focus on a subset of related research and industry efforts. Each section details a specific feature of the project as it related to our work. These features include overall system architecture, resource management, and interoperability.

2.1 System Architecture

From the user's perspective, the application is seamlessly operating in a heterogeneous environment, however, underneath there is a complex set of cooperating components. A system architecture is comprised of abstractions (concepts and models) and software infrastructure components to enable an Ambient Intelligent Environment. There are several related research efforts that propose system architectures.

One research effort known as one.world [12] supports pervasive computing by providing a framework for application development. It includes a set of services for discovery, migration, and check-pointing. Applications are organized in a component hierarchy and communicate through an asynchronous event passing mechanisms. All data and communication are structured as tuples passed between exported/imported event handlers (which may be statically and dynamically linked), supported by remote event passing (REP). one.world is based on the principles that the system should expose change (coming and going of devices and services) to the application, so that they can implement their own management strategies. They are worried that traditional RPC-like mechanisms hide change from applications. The second principle, encouraging ad-hoc composition, supports the notion that applications should support new behaviors, without changing the application itself. Finally, they advocate separating data and functionality, essentially an argument against object-

based architectures. This enables better sharing, searching, and filtering of data.

The main difference between this work and ours is the focus of our research on higher level services (content, capabilities, and personae) and management of change and heterogeneity through the resource manager as a system service. In one.world, applications are responsible for observing and managing change themselves through the offered lower-level services.

Metaglue [1, 21] is a programming language for building multi-agent systems. These multi-agent systems provide the software base for MIT's Intelligent Room Project. Metaglue extends Java by defining a new class as the root of all agents in the system. By extending this class, agents have access to Metaglue primitives and run in the Metaglue Virtual Machine. Primitives such as *tiedTo* (defines where the agent must run) and *reliesOn* (connects agents so they may request services of each other). System services include agent swapping, start-up, migration, and debugging. Agents themselves perform simple tasks, however, by providing services for other agents to utilize, agents can combine to perform complex applications.

Metaglue and our work shares the goals of having a clear programming model and set of services. Our work differs in the approach: They approach the problem form a programming language perspective by adding primitives to Java. Our work advocates a more general programming model of communicating tasks and is indifferent to the programming language used.

Another related research project is GaiaOS [22]. GaiaOS is a middleware operating system. It imposes a layer of indirection between the commodity operating system and the applications. The GaiaOS application framework extends the traditional model-view-controller model by adding a coordinator module. This module is responsible for managing the other three modules to support mobility, adaptation, and dynamic binding. GaiaOS abstracts the physical space and devices into a unified system which they call an *active space*. The GaiaOS kernel provides a variety services for use by applications: event management, presence detection of digital and physical entities, context of active space (useful for personalization), space repository to store information about hardware and software components, and the context file system (to personalize data access).

While the overall goals and system architecture of GaiaOS is similar to ours work, we focus less on the human-computer interaction (HCI) aspect of ubiquitous computing, and more on leveraging multi-media and sensor networks. We are interested not only in single office-like applications, but cooperation and co-optimization of many applications.

2.2 Resource Management

An application cannot be customized to a particular environment because people and devices are constantly coming and going. Therefore, the system must map the application to the environment at run-time. This requires resource management. Typically, resource management maps abstract functionality to concrete devices; along the way the abstract functionality may be refined, allocated, arbitrated, and scheduled. Here we discuss aspects of projects, some of which were discussed earlier, that correspond to resource management.

Metaglue [1] has a notion of resource mapping described in [8, 9]. Agents can refer to each other through abstract capabilities. This reference is made through the *reliesOn* primitive using specific naming conventions. This abstract capability is then mapped to a concrete device. The resource manager is centralized; it first performs a service mapping by weighing relative cost and utility. The output of this process are candidates for mapping. Then a constraint satisfaction engine is used to arbitrate between requests for optimal resource assignment.

This work closely resembles our own, in that they desire to make it part of the underlying service infrastructure. Our work is also similar to theirs in that we both advocate using high-level notion of device functionality to be mapped to concrete devices. Our work differs in how we specify application functionality (our use of flow-graph abstraction versus the *reliesOn* primitive). Furthermore, we extend the notion of resource allocation and arbitration in the resource manager, to include interoperability (*e.g.* matching types of stream endpoints by inserting transforms).

GaiaOS [22] also has a notion of high-level applications which are mapped onto the current active space. Applications are described in an active space independent manner by the developer. This description includes the application components, minimum and maximum instances allowed, and dependencies (*e.g.* the application requires audio output). This description is used along with the target active space's repository service to build a concrete application description for execution.

GaiaOS is similar in that we both advocate describing an application in terms of abstract capabilities and constraints. However, their notion of resource mapping is very limited. First, the environment may change during run-time while their resource mapping is done statically before application start-up. There is no run-time adaptation. Furthermore, there appears to always be a one-to-one mapping between abstract functionality and concrete device, where in reality there may be a one-to-many or many-to-one. Finally, issues of arbitration and scheduling are not addressed.

Network-integrated Multimedia Middleware (NMM) [16] enables audio/video streaming through fine-grained processing units represented as a graph. A graph of processing units is constructed by type matching unit ports. Units are capable of demuxing, decoding, and sinking audio/video streams. The graph is automatically mapped to physical resources, given constraints on bandwidth, QoS, etc.

This work resembles our goals for maintaining a set of constants (*e.g.* QoS) when mapping the application tasks to physical resources. It is different because it focus only on multi-media streaming and not other aspects of the AIE, such as sensor networks, mobility, etc.

2.3 Interoperability

Interoperability is the ability of any device to communicate with any other device despite differences in the means of communication. It is unlikely that all devices will share the same networking stack, in fact the diverse needs and constraints of the devices preclude this. As a result, there needs to be mechanisms, likely part of the resource mapping process, in which the networking stacks of communicating devices are bridged. This section reviews several efforts in this area, ranging in application from Internet services to ubiquitous computing.

With regard to applications in Internet services, Automatic Path Creation (APC) [18], for the Ninja Project [11], attempts to solve the device interoperability problem for mobile clients when data formats do not match. In addition, they may insert modules, such as forward error correction (FEC) or compression, to dynamically adapt to resource variations in the network. They perform logical (*i.e.* choosing modules) and physical mapping for transcoding needs between endpoints in large scale Internet services. Our work generalizes and extends the mapping approach beyond data formats to include additional layers in the network stack.

APC inspired other research efforts. Similar to APC is the project described in [7]. They attempt to solve some of the shortcomings of APC by using a dynamic programming based approach. Their work accounts for network link properties and node/link resource constraints, when attempting to bridge differing data formats. Also based on APC, is work called Paths [15]. They have made incremental improvements to APC to allow two endpoints in a ubiquitous computing environment to interoperate at the presentation layer (*i.e.* mismatch in data formats). Paths takes a mediated, infrastructure approach, similar to ours, however, it is still only focused on a single layer of the networking stack.

Universally Interoperable Core (UIC) [23] is a reflective middleware solution for ubiq-

uitous computing environments. UIC has been used in GaiaOS [22]. In UIC, controlled devices export their communication middleware properties which are used by the controller application executing on a controller device. The controller application dynamically modifies the controller device's communication middleware. There are several differences between our approach and UIC. While both perform dynamic reconfiguration to support interoperability, we dynamically compose communicating modules instead of directly modifying the device's communication middleware. Also, similar to other research efforts, this work is only interested in interoperability at a single layer of the networking stack.

Related to UIC, is another reflective middleware solution, ReMMoC [10]. The main differences between this and UIC is the extension to allow interoperability with different discovery protocols. Furthermore, ReMMoC allows different interaction paradigms (*e.g.* publish-subscribe and data-sharing), besides object-oriented request brokers (ORB) of UIC. This work shares the same key differences to our work that UIC does.

Mapping has also been investigated in the context of communication middleware (*e.g.* UPnP). [19] presents a bridging framework between a set of common communication middleware using a common semantic domain. They dynamically choose which translators to instantiate depending on the requirements of the participating devices. We are not only interested in dynamic mapping to bridge communication protocols, but also data formats (*e.g.* MPEG4 to H.264) and physical connectivity (*e.g.* bluetooth to 802.11).

An alternative to mapping is the the use of mobile code as [5, 6] have proposed. They use a low-level meta interface for generic discovery, data transfer and control. This interface allows a controlling device to "teach" a controlled device to interoperate, essentially extending their functionality. While this approach can overcome many of the differences in the communication protocols, it is still does not address differences in device data formats (*e.g.* presentation layer).

Universal Plug and Play, or UPnP [25], is a framework to support device interoperability across a network. UPnP defines devices, services, and control points; devices may contain services and nested devices. UPnP defines a protocol stack based on common Internet standards which enables addressing, discovery, description, control, eventing, and presentation of devices and their content. The Digital Living Network Alliance (DLNA) [3] builds upon UPnP by providing a set of audio/video interoperability guidelines. These guidelines include mandatory and optional multi-media formats and digital rights management. DLNA audio/video devices are broken down into three categories: home network device, mobile handheld device, and infrastructure device. Infrastructure devices provide bridging between physical layers and computational resources for transforming between

Table 2.1: Comparison of related work.

Solution	System Arch.	Resource Mgmt.	Interop.
one.world	×		
GaiaOS	×	×	
Metaglue	×	×	
UPnP			×
NMM		×	×
APC			×
Paths			×
UIC			×
ReMMoC			×
uMiddle			×
Obje			×

media formats/protocols. Unfortunately, UPnP and DLNA still limit the set of standards and offer no dynamic or flexible approach to adding new devices and standards.

Table 2.1 shows a table comparing the related work. The table shows which related works provide a system architecture, resource management (mapping) strategy, and interoperability. Our goal is to provide a solution which addresses each of these aspects.

Chapter 3

System Architecture

The convergence of different applications, devices, and networks in the home presents interesting research challenges. This section addresses several of those challenges by defining a set of abstractions, an application model, and required services for the Ambient Intelligent Environment (AIE). Our system architecture consists of applications, high-level services (personae, content, and capabilities discovery) and low-level services (resource mapping, geometry, security, I/O, process management, etc.), a hardware abstraction layer, and a network of distributed, heterogeneous devices. Figure 3.1 shows a layered representation of our AIE system architecture. Each of these elements will be discussed below.

3.1 Abstractions

The *environment* is an umbrella abstraction. The environment is a grouping which contains virtually every piece of data or device(s) at the user's disposal. For example, the environment contains:

- consumer electronics (*e.g.* DVD player, computer, hand-held tablet PC, cellphone, television, stereo),
- sensors/actuators (*e.g.* motion, light, temperature),
- appliances (*e.g.* heating/cooling system, lighting, cameras),
- subscription services/devices (*e.g.* television channels, Internet, security, power, telephony),

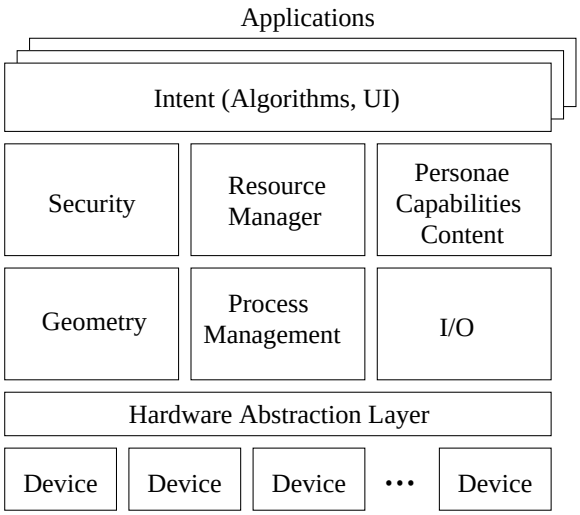


Figure 3.1: Layered diagram of our AIE system architecture.

- infrastructure (*e.g.* disk storage, routers, bridges, wireless access points, compute nodes),
- content (stored and live content: stored – *e.g.* camping pictures, home movies, calendar, email contacts, etc.; live – *e.g.* security camera, television program, radio broadcast, current temperature, etc.)
- and people or organizations who may make use of the elements (*e.g.* you, me, guests).

The environment in its entirety is too overwhelming for any person or algorithm to use efficiently. A subset of this environment, which coexists and interacts in some scope for some purpose, forms the *current environment*. The current environment is the foundation upon which services (see Section 3.2) can be built. The questions such services can answer may include: "what is user X's position?" "what devices exist?" "what movies are available?" "which device at my current position renders video?" "what is the temperature?" etc. These services can also notify of events such as "user X has changed location," "a new device is available," "a movie is no longer available," "the temperature has reached X," etc.

Scoping is applied to the environment to yield the current environment. The scope can be spatial, temporal, or based on meta-data. The environment scope may include simple and familiar concepts such as "my house," "my room," or the "living room," but may also include "two rooms on the opposite sides of a planet" with participants of a video-conference. In the scenario from Section 1.2, the scope is the current room where John and Adam are

located.

The current environment may be further broken down into abstractions known as *personae*, *capabilities*, and *content*. These elements are elaborated in the following discussion.

3.1.1 Personae

The importance of the persona is to capture user-specific details that affect the interaction between content and capabilities, thus enabling multi-user optimality. A persona may represent a single person, groups of people, or organizations (*e.g.* fire department). A persona contains preferences, permissions, and properties. Preferences represent personal choices captured in some meaningful way to the system. In addition, a persona with sufficient privilege may set rules of operation for the environment (*e.g.* parent limits speaker volume to *X* in home). In essence their virtual presence is enough to enforce their preferences. Permissions are user's access rights to devices and content, based on content attributes and meta-data. For example, permissions may limit a user's ability to watch rated-R movies or control security related devices. Finally, a persona's properties contain information that describe the represented user(s), *e.g.* birth date, passwords, and other data. The run-time system interprets and reacts to user's interactions based on user's persona and resolves conflicts between multiple personae.

3.1.2 Capabilities

Capabilities uniformly capture physical resources (*e.g.* devices, network, programs) in the environment. The resources are exposed as a capability by the functionality they provide (*e.g.* radio – *audio creator*, television – *video renderer*, temperature sensor – *"current temperature" content source*, router – *connection from A to B*, transcoder – *conversion from format X to Y*). Capabilities may abstract familiar devices such as: set-top box, DVD player, remote control, stereo system, security system, lighting and appliances, energy monitors, as well as the unfamiliar: sensor arrays, hidden compute devices, smart materials, and contextual control objects.

The scope that defines the current environment, sets spatial and temporal conditions for availability of certain capabilities. A uniform description language describes each capability for all devices in the AIE. Integrating a new device simply reduces to exporting a set of capabilities using this language, which the system automatically recognizes and uses appropriately (more detail may be found in Section 3.4). Unrecognized capabilities may be may be "understood" by the system through an online database. By using this scheme, the

run-time system enables simple integration of new devices and therefore zero-configuration in ZUMA.

3.1.3 Content

While capabilities abstract the physical world, content abstracts information that capabilities can manipulate. Content may represent a range of information, either live or stored: media streams, sensor readings (light, temperature, motion, identification), security information, energy monitoring results, etc.

Content and capabilities share a uniform typing system as part of the previously mentioned description language. This allows for consistent treatment of content and capabilities, and trivial integration of new content. For example, classifying a video as MPEG-4 and a television input as MPEG-4, simplifies their association or compatibility when user request is implemented. Integration of new content only requires a compatible capability using the description language. These concepts enable zero-configuration. While both content and capabilities find common ground under this language, a full exploration of its syntax and semantics is the subject of future research.

In some cases, content and capability represent a common physical resource. For instance, a capability to play a video and the content from a DVD track, are provided by the DVD player. Similarly, a temperature sensor provides a capability to emit temperature and the "temperature" content. Although these examples may suggest merging capability and content into a single abstraction, they are not always one-to-one. For example, devices that transform (*e.g.* picture-in-picture) or render (*e.g.* speakers and video screen) provide capabilities to operate on a range of content. Furthermore, many capabilities may be aggregated to operate on or produce a single piece of content (*e.g.* all temperature sensors in a room provide an average room temperature).

3.2 Services

The environment abstraction is the foundation upon which services may be built. These services are used by active entities such as applications and other services. For example, in our scenario the application uses discovery to find capabilities (*e.g.* video rendering), content (*e.g.* set-top box content, DVD tracks), and personae (*e.g.* John and Adam) in the room. The low-level run-time system needs information about devices (*e.g.* television, DVD player) and user locations. To accommodate these needs, services such as resource

management, discovery, security, process management, geometry, and input/output (I/O) must exist. Each of these will be briefly discussed below. The focus of this research is on the resource manager service (discussed further in Section 3.4) as it represents a unique aspect of our research.

3.2.1 Resource Manager

In conjunction with other system services, the goal of the resource manager is enable all tenants of ZUMA. Applications are specified abstractly allowing them to be environment-independent (more discussion in Section 3.3). The resource manager maps the application to the current environment by first refining the application's requirements, then allocating and arbitrating for physical devices.

3.2.2 Personae, Capabilities, and Content Discovery

Parts of the environment abstraction manifest as discovery services to which applications can post queries and register subscriptions for events. Scoping is applied through the discovery interface to yield the current environment.

3.2.3 Security

This system service is responsible for the authentication of users and devices. Furthermore, it is tasked with managing the policy enforced for maintaining user privacy at all levels of the networking stack.

3.2.4 Process Management

Migration and replication of application processes require careful management of a distributed set of devices. This service is responsible for properly collecting and moving active processes at the request of the application or resource manager.

3.2.5 Geometry

Inherent to an AIE is the localization and identification of not only users, but also the devices which populate the environment. Geometry is a combination of techniques in computer vision, sensor networks, RFID, etc. to enable awareness of the physical spaces.

3.2.6 Input/Output

Applications must have the ability to upload and download state to and from the infrastructure in a consistent and safe manner. This system service is responsible for managing persistent and reliable storage over a distributed set of devices.

3.3 Application

An application is a description of functionality. When executed, the application seamlessly utilizes a set of devices in the environment to accomplish a goal. Example home applications include multi-media viewing, home automation, monitoring, and security. Applications have many interesting properties depending on their goal(s):

- When an application requires user input, or for that matter, produces output for the user's consumption, this interface to the user must be multi-modal or capable of runtime reconfiguration. A user's input may be captured as sound, touch, motion, presence, lack of presence, gesture, etc. Similarly, the output of an application may be in the form of any combination of the five senses. Therefore, the application's user interface must be decoupled from the actual application processing.

- An application will likely be distributed – its execution will take place on several devices connected through a networking fabric.

- Applications may be either ephemeral or long-running. Consider a security application. It may run continuously for years while capturing video, audio, and other sensing data. Ephemeral applications, such as audio-video, may only last a few minutes, or few hours, ultimately to be shut down and eventually re-started.

- Application start-up occurs within a "shell" which is part of the AIE. This shell may be presented graphically as well. The shell is akin to the command line shell many people are familiar with with today's UNIX or Linux systems. This shell is a special program run as part of the environment allowing a user to start, stop, pause, and kill applications.

3.3.1 Programming Model

The application describes the user interface and necessary callbacks to handle user input. The application's core defines how it's state will be structured and manipulated, and utilizes

the services of the environment to build and maintain its state.

Conceptually, an application can the thought of as a *flow-graph*, or *network of tasks*. The application flow-graph is a directed graph $G(V,E)$, where V is composed of three general *task* types: producers, consumers, and transforms. E is a set of edges representing unidirectional byte streams (may be fully or semi-structured) which capture the communication between task nodes. Byte streams are for data and control. A producer sources one or more byte streams; a consumer sinks one or more byte streams; and a transform converts one or more input byte stream into one or more output byte stream. Typical examples of producers, consumers, and transforms (both a consumer and producer):

- *Producer*: persistent storage, sensor, camera, set-top box.
- *Consumer*: actuator, video/audio renderer.
- *Transform*: picture-in-picture, re-size, transcoding, encryption.

A typical application will have a consumer node for rendering the user interface, a transform node which will convert user inputs into meaningful programmatic callbacks, and a transform node which maintains the application data and interaction with services (*i.e.* the application's core). The application may generate, using services, local state which augments this basic flow-graph with other capabilities. For example, an application which streams a movie will query a discovery service for a content source and renderer (producer and consumer). These objects will be used to augment the flow-graph with a media stream between the new producer and consumer. Essentially, the application will modify this flow-graph at run-time.

3.3.2 Execution Model

Each task in the network of tasks is single or multi-threaded. A task may also be a hardware module wrapped in a software layer. Tasks will be allocated to the resources of the network in such a manner to meet user and performance constraints. Examples of resources include link bandwidth, memory, CPU cycles, and reconfigurable circuit fabric slices. The communication between tasks occurs over the network through byte streams. This communication consists of command/control message passing and content streams. All tasks will run in parallel. Message passing may be short-lived to optionally initiate or complete long-lived content streams. Since tasks may be written by third parties, it is expected that communication may not always be feasible between tasks. Also, the devices upon which

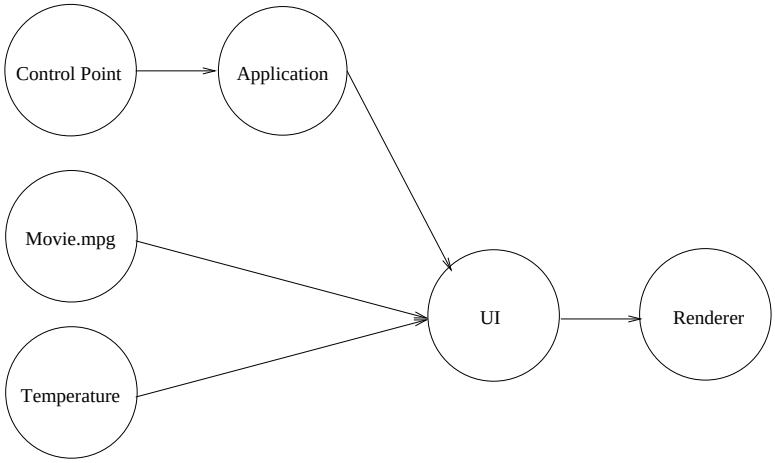


Figure 3.2: Simple application which streams a video to a local renderer and monitors local temperature.

software tasks are run may not be capable of communicating due to protocol or data format mis-match. For example, a consumer task may require MPEG-4 formatted video, however, the producer task can only source H.264 formatted video. Therefore, the appropriate mechanisms to enable interoperability between tasks will be automatically instantiated on the resources of the network (for more information see Section 3.4).

3.3.3 Example and Discussion

A simple flow-graph which streams a video to a local renderer and monitors local temperature is shown in Figure 3.2. The application flow-graph contains two producer nodes each sourcing content byte streams of "Movie.mpg" and "Temperature." The "Application" and "UI" Transform nodes implement the user processes for the control algorithms and user interface (some uniform graphical language such as HTML or SMIL). The "Control Point" node is another producer which sources an event byte stream sending control messages from the user to the application's control algorithms. Finally the "Renderer" node sinks the output of the UI node to display the streaming movie and the overlayed temperature text.

Notice that the nodes in the flow-graph of Figure 3.2 represent capabilities and content. These are obtained through the discovery service. The reasoning for requiring applications to specify their intent in terms of the capability abstractions is the following: As the AIE evolves, new devices and functionality will be introduced on a regular basis, while applications should remain relatively constant. Moreover, the diversity of device functionality in a

single AIE will be great and applications will often require resources dependent on context or location (*i.e.* the living room may have an HDTV cable video screen, while the bedroom video screen is a legacy device). This requires applications to specify their needs through a uniform, but abstract interface, thus allowing the system to determine the appropriate device to be used and evolve with future upgrades.

The construction of a flow-graph by an application may look like the following pseudo-code:

```
Scope s = Scope.localSpace;
CapabilityList l;
l = CapabilityService.find(s, "video renderer");
CapabilityNode renderer;
renderer = new RendererNode(l.first());
CapabilityNode ui = new UINode();
...
ContentNode movie = new ContentNode("Movie.mpg");

Netlist n;
n.add(renderer);
n.add(ui);
n.add(movie);
...
n.add(application);

n.connect(movie,ui);
n.connect(application,ui);
...
n.connect(ui,renderer);
```

Note the user is not concerned with where an application is executing (*i.e.* on which physical device). An application should be independent of the control point, such as keyboard or touch-screen, and independent of rendering devices, such as a video monitor or speakers. The application will include abstract nodes for control points and rendering and be capable of multi-modal user interaction.

Once the flow-graph has been specified, constraints are given explicitly by the application. Some of these constraints are already specified in the pseudo-code above, such as

`Scope.localSpace`. This requires that the discovery services only include the local space around the user when searching for video renderer capabilities. Other constraints such as QoS, conflict resolution, etc. can be appended to the netlist specification of the flow-graph.

Finally, the flow-graph is ready to submit to the resource manager service. The resource manager service is responsible for taking the high-level description of the application functionality and mapping it into the set of devices available in the AIE given explicit application constraints and implicit user constraints (*e.g.* minors do not have permission to watch certain content). Once the resource manager performs its job, it returns to the application a set of handles for devices which are properly configured and networked to perform the specified user intent; otherwise the resource manager returns a failure condition. Upon successful mapping, the application can interface to the devices through the hardware abstraction layer (uniformly API for device capabilities) and control each device as it needs. If there is a change that requires action by the resource manager (*i.e.* the user changed position and requires a different "local" video renderer), then it will suspend access to those devices through their handles, re-map the necessary nodes, and return control the application.

3.4 Resource Manager: Run-time Refinement, Allocation, and Adaptation

The resource manager transforms the task graph, through refinement, allocation, and adaptation, into a graph which abides by the constraints specified by the application and can execute in the current AIE.

The resource manager is a logically continuous process: it constantly re-maps the required tasks to reflect any changes in the tasks or the current environment. The match between the requested elements of the task and the devices will be exact (*i.e.* legal), otherwise the mapping process fails. Of course, legal mappings are dependent on the constraints of the application and the persona's preferences: a persona may accept degraded video quality when transferring a video session from the HD-TV to a handheld TV or a task may be amenable to time-multiplexing (*e.g.* of a compute node for non-real-time elements of a task) if resources are scarce. Through callbacks, the manager service provides feedback to the application to indicate success, alert of changes to the environment that might be of interest, or indicate a failure to meet a constraint.

Although the discussion of resource manager may imply a centralized mapping process, a *distributed* implementation would likely be a preferable, scalable approach. Distinct geographically and temporally delineated environments, people's preferences, permissions,

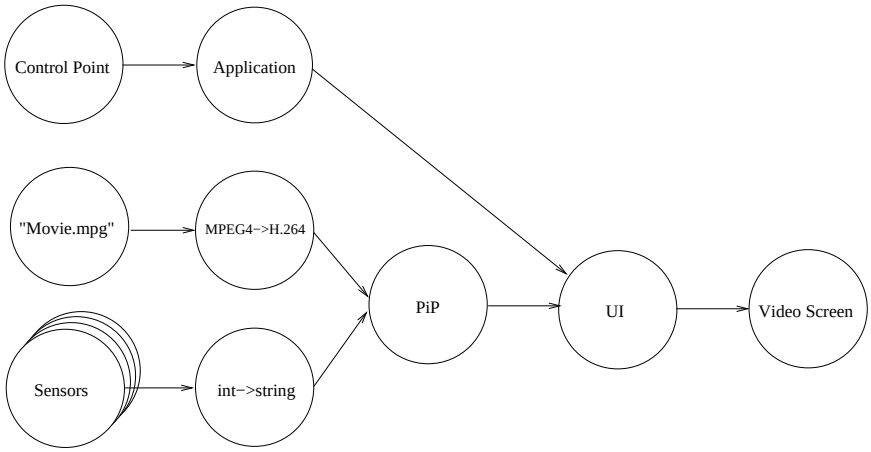


Figure 3.3: Refinement of simple application shown in Figure 3.2

and priorities naturally subset resources, content, and personae that are involved in a given mapping decision. These non-intersecting subsets enable us to parallelize and distribute management across a range of compute devices. A scenario where a single mapping decision involves all or nearly all content, devices, and personae will be rare in practice.

Upon receiving the flow-graph specification from the application, the resource manager must perform the following steps to map the flow-graph into a realizable implementation: refinement, resource allocation, and adaptation.

3.4.1 Refinement

Once the flow-graph has been specified, the nodes of the graph and the links between the nodes may require refinement. First, refinement maps the abstract nodes of the application task graph to actual functionality the current AIE can provide. This is done by using a table which maps capabilities to potential devices. By comparing Figure 3.2 and Figure 3.3, the “Renderer” node is refined to a “Video Screen” node. Once these device are known, refinement must elaborate the links between devices to satisfy application constraints, such as data format matching, QoS, etc. For example, a device may only be capable of producing a movie file in a certain data format which is incompatible with the consuming device. In this case, the proper solution would require a transcoder (transform node) capability to be inserted between the producer and consumer. In Figure 3.3, a transcoder nodes was required to convert ”Movie.mpg” from MPEG-4 to H.264 format.

Furthermore, mismatches in byte stream fan-in and fan-out may also have to be solved by inserting a funnel or horn transform node. For example, a picture-in-picture node (PiP)

was inserted between the two producer capabilities in Figure 3.3 for input to the UI capability node.

If an application requires a resource that is currently occupied by another application, then depending on the priority of the applications and the sharing defaults, the two applications may share the device through the instantiation of another funnel transform capability (*e.g.* split-screen for a video renderer) or would require migration and possibly re-allocation of a resource for the evicted application. In Figure 3.4, two applications share a single video renderer due to both flow-graphs requiring the "local" rendering device. This example also shows how the user experience may be enhanced when applications cooperate in a seamless manner.

3.4.2 Resource Allocation

Next, the resource manager performs allocation and arbitration of refined nodes (their functionality) to available resources (devices and network bandwidth). Resource allocation is concerned with set of costs amongst users depending on ownership, permissions, and priorities. Some capabilities will be completely specified, for example, the application may specify the exact video rendering device to use. In other cases, the rendering device may be dependent on the user's location (*i.e.* the "local" rendering device). Allocation of a particular device to a higher priority user may require migrating a process for, or re-allocating resources to, the lower priority user.

Depending on the set of devices available in the current environment, a refined flow graph may contain several nodes which will actually "run" on a single device. For example, in Figure 3.3 the UI, Video Screen, and PiP nodes may all be allocated to an advanced television capable of receiving multiple streams and conforming to the UI layout. Alternatively, compute nodes may implement the transcoding nodes for each stream and the PiP node, sending a single composed stream to a less capable television.

3.4.3 Adaptation

When the change occurs in the AIE, such as a new device coming online or leaving, a user moving to a different location, etc. the resource manager is responsible for reacting to this change within the scope of the application's flow-graph constraints. When a user enters the room, they may have additional content available on a mobile device they are carrying and depending on how the application specified the capability constraints in the original flow-graph, the mapped flow-graph may change. For example, the most "local" control

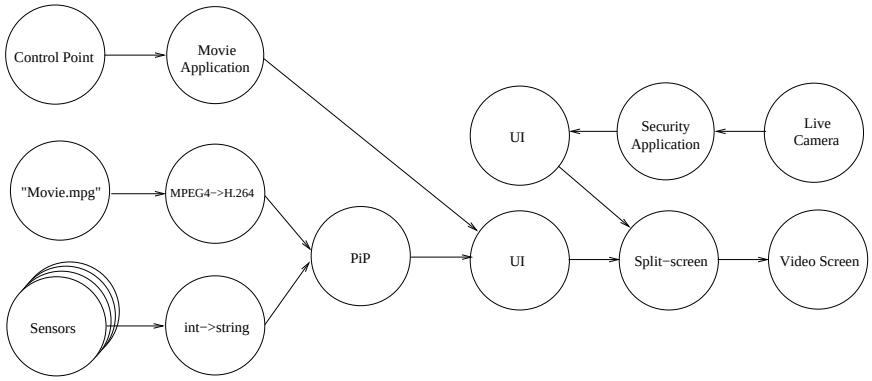


Figure 3.4: Refinement of two simple applications sharing a resource.

point may now be the cell-phone in the person's pocket, rather than the keyboard from a laptop. Similarly, when the user moves to a different location, or changes the constraints for mapping (*e.g.* downgrades the required video quality), the manager may replace a transcoding node with another type, or free some resources previously consumed by an expensive QoS reservation. All of these actions taken by the resource management service seek to simplify the application functionality, as well as, make more effective use of the AIE resources. Essentially, adaptation boils down to the resource manager performing refinement and allocation over time.

3.4.4 Execution Platform

The execution platform consists of a hardware abstraction layer and physical resources of the AIE.

3.4.4.1 Hardware Abstraction Layer

The hardware abstraction layer (HAL) consists of low-level, uniform interfaces for device discovery[1] and control of physical resources. Some devices may already have their own HAL in the form of communication middleware (*e.g.* UPnP). In this case, the resource manager would be responsible for bridging the discovery and control protocols of those devices to the application. Discovery is a process by which devices upon start-up will broadcast their presence and capabilities to the network. Depending on the implementation, this information may be stored in a repository for future access, or future uses of the

[1]This is a lower-level discovery than the services described above.

capabilities may require on-demand discovery broadcasts. In this situation, the device will reply with its capabilities on a case-by-case basis.

Once the devices have been discovered, their interfaces are known and they can be controlled. Typically, the control will be sourced from an application. Since an application cannot possibly support an unknown interface, a uniform control interface is provided for different types of capabilities. For example, a renderer capability will have a specific interface for receiving multi-media streams, setting state variables, and reading state variables.

3.4.4.2 Resources

Resources are the physical devices upon which applications and services of the AIE operate, including cell phones, set-top boxes, sensors, lights, actuators, televisions, IR remote controls, PDAs, stereos, speakers, cameras, etc. Some are obviously mobile, some are not. Their roles as part of an application or service will be determined by the application and its current mapping by the resource manager.

Chapter 4

Implementation: "Home-Gateway Demo"

While Chapter 3 illustrated the important concepts of our system architecture for an Ambient Intelligent Environment (AIE), this chapter will illustrate how several of those ideas have been implemented in what has become to be known as the "Home-Gateway Demo."

4.1 Introduction

This section will give a short introduction to the implementation, its history, and the assumptions we were working with during the demo's most recent incarnation.

4.1.1 Brief History

The demo idea was originally conceived as a design driver for the Gigascale Systems Research Center (GSRC) [13]. A design driver is intended as a concrete driver for research projects. This design driver's application was the "smart-home," specifically distributed media streaming, transcoding, and flexible control. The Home Gateway consists of a central framework which orchestrates routing, instantiation of run-time transcoding, and manages state. Various groups would contribute results of their work, essentially plugging them in to the framework we created.

At the beginning of summer 2005 several grad students, along with Professor John Wawrzynek, started designing the framework. This effort was led by graduate student Yury Markovsky. Our goal was to have streaming audio and video between a source and sink by the end of the summer. Unfortunately, due to technical problems with a hardware MPEG-4

decoder, we were only able to show audio streaming and a photo slide-show. This demo consisted of two media sources, two media sinks (*i.e.* one slide-show sink, one audio sink), a central server, and a wireless remote control running the media streaming application. On September 9th, 2005 we demonstrated this first effort at the annual GSRC symposium in San Jose, California.

As with any first systems project, the second project is more ambitious. After the September demo, we also began research discussion that would eventually produce the work in Chapter 3. During this time we began formulating the goals for our next demo. Top on our list was not only video streaming, but video transcoding. We wanted to demonstrate the concepts of the resource manager with some simple constraints (*e.g.* location of the user). We also wanted to develop a more robust device management framework, bring everything under one code-base, and integrate a small sensor network. For the GSRC research symposium on March 14th, 2006 in Berkeley, California, we demonstrated video streaming with transcoding (between MPEG-4 and Motion-JPEG), adaptability (when the user changed location between video sinks) with a subset of the resource manager (which ran as part of the server), a more general and robust device management framework, and support of multiple multi-media streams (audio, video, and slide-show).

4.1.2 Assumptions

In designing the current demo we made several assumptions:

1. Smart-home tasks and technologies – Our work would focus on controlling parts of home environment, and therefore, hardware reconfiguration (using FPGAs) is key. An application must be capable of run-time reconfiguration because a home environment is very dynamic. This motivated our focus on the resource management services and a more robust discovery/control protocol.

2. Convergence of personal computers, consumer electronics, and sensor networks – A smart-home environment must accommodate this convergence and must support the heterogeneous network of devices that accompanies this convergence. What is good for personal computers may not be the best for consumer electronics or sensor networks. For example, common networking technologies between the three types of electronics is almost non-existent. Furthermore, they need to be treated the same under one framework. This framework must be general enough to capture all the necessary details of these divergent technologies.

3. Interoperability – To simplify the development process we made several assumptions about the interoperability between devices (as a research subject this is left to future work):

   (a) Common kernel – All devices run a common device discovery and control kernel. The kernel is a small C library to which every client and service is programmed against to facilitate communication through an application overlay. This kernel handles server control and discovery of clients, and the control of the server and clients by the user.

   (b) Common Internet protocols – Underneath the common kernel, we utilize common Internet protocol standards, such as TCP/UDP, IP, Ethernet, and 802.11g. Treating all devices the same on an IP network greatly simplified the development process.

   (c) Only presentation conversion – Only content data format mis-matches were handled with regard to interoperability. These mis-matches were handled by automatic recognition and inserting the necessary conversion module between the source and the sink. Instead of the data from the source being streaming directly to the sink, the data is routed by the server through this transcoder, and then to the sink. This routing is established at run-time.

   (d) Wrapping of hardware resources – Some resources, such as transcoders (chain of decoder, then encoder), are implemented in hardware on Field Programmable Gate Arrays (FPGAs). Their integration into the rest of the system is accomplished by wrapping this hardware module in a software layer. This allows this module to be treated no differently than other devices in the network.

4.1.3 Demo Overview

We have developed a prototype to validate several concepts of the proposed architecture. Although a full implementation of the system architecture proposed in Chapter 3 is beyond a scope of an academic project, the prototype includes all basic features and comprises a compute and routing hub, video and audio clients, media storage devices, remote control and sensor devices (see Figure 4.1). These devices share a common IP network, supported by the wired and wireless connections.

The content sources consist of storage devices that stream audio and video content upon request. The sinks are video and audio clients that decode and render incoming streams.

The decoding is performed either in software or in FPGA hardware depending on the real-time computational requirements.

A compute node, either an FPGA or a processor, is a special device managed by the run-time system. Field Programmable Gate Arrays (FPGAs) are post-fabrication programmable integrated circuits that contain microprocessor cores (ideal for low performance, complex control) and configurable logic (ideal for high performance, minimal control computations). The run-time system dynamically programs the compute nodes to execute a particular operation. For example, if the source content format does not match the sink format (*e.g.* Motion-JPEG video source and MPEG-4 video renderer), the run-time system allocates an FPGA and programs it to trans-code Motion-JPEG into MPEG-4 in real time as the video is streamed from its source to destination.

These programmable compute nodes are available on the BEE2 FPGA board [4], which contains five FPGA chips and has several functions in our prototype. The main FPGA runs the server (a subset of architecture) on Linux. The main FPGA also serves as the router between the four slave FPGAs and external IP network. The slave FPGA are configured at run time with trans-coders.

The prototype has two small "human presence" sensors that communicate through sensor-network motes (running TinyOS [24]) to the server. They are implemented using floor pressure mats.

To summarize, the Home Gateway demo consists of the following features:

- Wireless remote control with graphical user-interface.
- Content sources serving audio, video, and photographs.
- Multiple multi-media sinks for audio, video, and photographs.
- Discovery of all devices and network resources (*e.g.* transcoders).
- Common device control protocol for all devices.
- User locationing with sensor network motes and pressure mats.
- Video streaming in MPEG-4 and Motion-JPEG formats.
- Audio streaming in MP3 format.
- Streaming over wired and wireless networks.
- Use of FPGAs for computationally intensive transcoding capabilities.

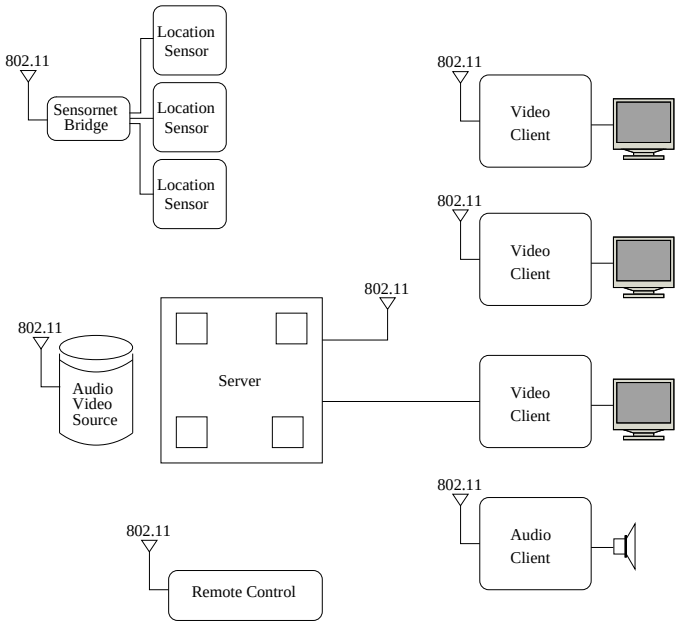


Figure 4.1: The top level architecture of the Home Gateway demo.

- Run-time adaptability (changing of streaming sink and possible insertion of transcoding support) of video stream based on user location.

4.2 Device Control Protocol

The device control protocol, or DCP, is the common framework we use to integrate devices together. Each device utilizes a DCP interface to facilitate discovery of that device, and subsequent command and query processing. The DCP framework performs the following tasks:

- defines a message format for discovery and session control,
- enumerates command and query types, their message layouts, and argument types,
- and a common library for use by clients for command and query construction, parsing, and error checking.

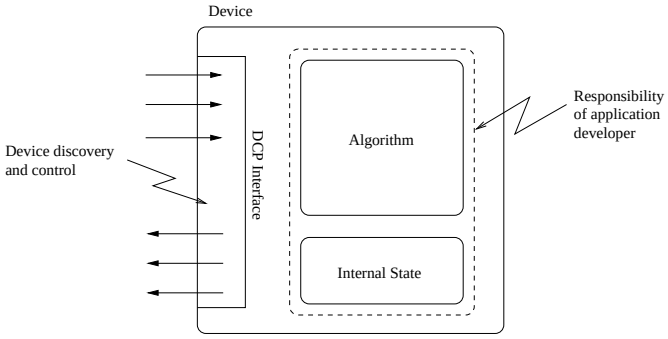


Figure 4.2: Device model consists of the DCP interface (discovery and control), and the developer-provided algorithm and management of internal state.

4.2.1 Device Model

Our device model is very simple. A device is modeled as an algorithm and internal state which utilizes the DCP interface for device discovery and control. For example, the algorithm could maintain a user interface and respond to user events; similarly, it could be a content source which responds to DCP commands by sending a media stream to a destination. Figure 4.2 illustrates this device model.

4.2.2 Communication Model

The communication model of DCP is a *server-mediated* model. All clients communicate through the server; it is the central repository for DCP state. In some cases, clients can initiate, via the server, direct communication in the form of streaming data between other clients. There are three types of communication in this model (as illustrated in Figure 4.3):

1. Single client to server (1) – In this situation, a client can initiate communication with the server or visa versa. This is useful for querying operation between the client and server, registration during discovery, etc.

2. Client to server, to client (2) – A client can cause the server to affect another client. This is typical of the remote control when it causes some state to be set in another client (before data is streamed).

3. Client to client (3) – Finally, a client can transfer data directly from itself to another client. This is always directed by the server, that is, the server will initiate a route between the source and sink. This is the model when data is streamed (*e.g.* video

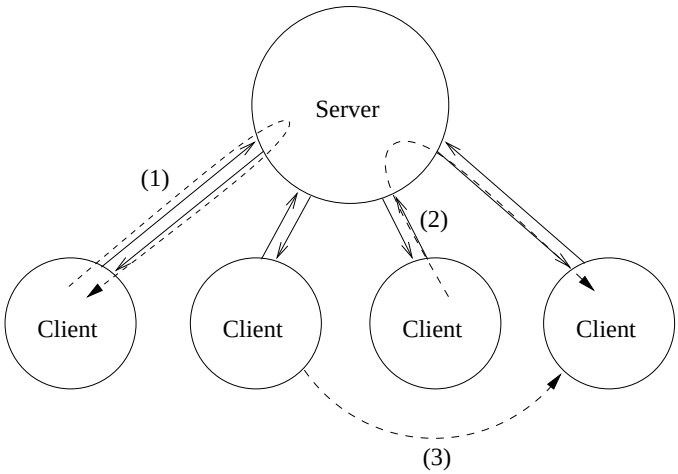


Figure 4.3: The communication model is a typical client-server model. (1) Devices can query, or be queries by the server. (2) They can initiate commands and cause changes in another client, and (3) can initiate communication (streams) between clients (after communication with the server)

playback). Multi-media streams between a source client and a sink client is characterized by a *3-box push* semantics: a client initiates a data push between a source client and a sink client.

The communication between a client and server is in a message format defined by the DCP. For each client, the server maintains state about the client and a TCP socket to the client. Messages are generated locally, passed over this socket, and processed by the receiver. The generation of messages is event driven; they are processed in order by the receiver. Processing a message may in turn generate another message in reply to the originating client, or to some other client. Events which initiate communication typically come from a device entering or leaving the network, an application request, or periodic timer events.

4.2.3 DCP Commands

This section describes common DCP commands, their associated message formats, and arguments. This discussion will focus their usage in discovery (Section 4.2.3.1) and session management (Section 4.2.3.2). For additional commands, we refer the interested reader to Appendix A.

The DCP command has the general format shown in Figure 4.4. The message begins

Figure 4.4: General DCP message format.

Table 4.1: DCP capability descriptions.

Capability	Description
`content_list`	A list of strings representing content file names.
`filename`	A filename to stream.
`destination`	A stream destination (push semantics).
`play`	Begin streaming to destination.
`pause`	Pause and hold position of stream.
`stop`	Stop and return to beginning of stream.
`next`	Quit current stream and begin streaming next content.
`previous`	Quit current stream and begin streaming previous content.
`volume`	Change the volume of audio output.
`brightness`	Change the brightness of video output.
`configuration`	Set the bit-stream configuration for a compute device.
`turnoff`	Turn off the client device.
`intype`	Used to set the expected media type for a specific port.

with the command, followed by the type, and ending with the command body. A set of common commands will be discussed further below. The type field is of either `ack`, `request`, or `error`. The `request` type is for messages originated by the client or server, while the `ack` type is for replies to a `request` type. In most cases the `ack` message type means the message is acknowledging the receipt of the `request` and optionally piggybacking data for consumption by the original sender. The `error` type is for replying to a `request` type message when there is an error. The message body typically contains arguments specific to the particular command and type. Several command message bodies contain values for device *capabilities* and device *type(s)*. Device types are different from message types. These capabilities and types are given in Table 4.1 and Table 4.2. Section 4.2.3.1 and Section 4.2.3.2 will give specific examples.

The DCP commands are sent and processed as strings, rather than building and packing bytes. This was done for ease of implementation. The DCP library contains a simple parser (implemented with Flex and Bison) to convert strings to message structs (see the grammar in Appendix A). The DCP library also contains functions to convert from message structs

Table 4.2: DCP device type descriptions.

Capability	Description
src	Device that sources content.
sink	Device that consumes content.
rc	Device that executes application to connect sources and sinks.
xcoder	Device that transforms media types.
sensor	Device that senses user location.
compute	Device that may be configured at run-time.

to strings. Another advantage of using strings is that applications such as the remote control can be easily scripted to automate startup. Further, logging activity in the server or clients is greatly simplified because commands are already in string form.

4.2.3.1 Discovery

Device discovery is performed when a device enters the network. Upon boot-up, the device will run the kernel designed for that device and the first thing that kernel will do is perform discovery (also known as registration). This involves contacting the server at a known IP address and port, and sending a `register` command. This `register` command contains information about the device. The server will parse this message and reply to the device with its assigned identification number. Depending on the information sent by the device, the server may generate several other messages which probe the device for more information. For example, if the device can source content, the server will ask for the list of content. Figure 4.5 illustrates this process. The following command messages may be used during the discovery process. For each command and message type, the arguments are discussed along with a general description detailing command's usage. For brevity, the `error` message type is excluded. It's message body is always the same for every message command: it contains fields for a `error-id` and a `error-message` which are processed as appropriate by the receiver.

- `register` – This is the initial communication between any client and the server.
  - `request`
    - Capabilities – Specific functionality offered by the device, such as providing a content list, a filename, play/pause/stop/next/previous, turn-off,

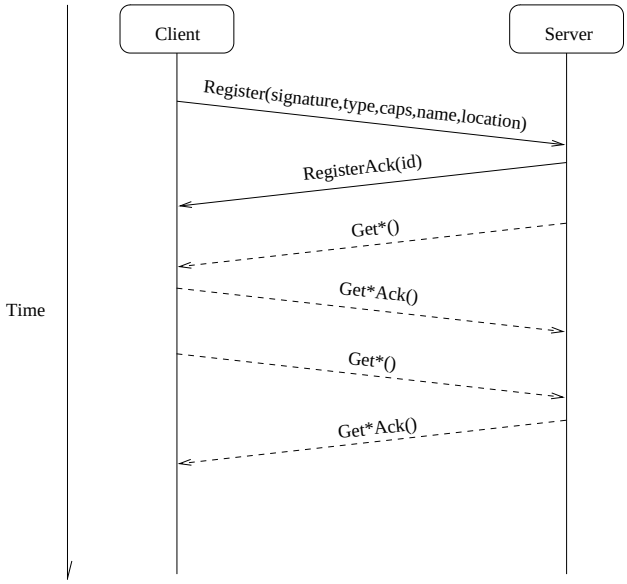


Figure 4.5: The DCP discovery mechanism initiated by the device's registration message and ack, followed by a set of optional (dashed arrows) queries to/from the server.

volume, etc. See Table 4.1.

  - Device types – A device can be of one or more types: source, sink, remote-control, transcoder, etc. See Table 4.2.
  - Port signature – A device may have one or more signatures which specifies a virtual port number and the type of media it can accept on that port (*e.g.* MPEG-4, text, etc.).
  - Location – The location of the device.
  - Name – The textual representation of the device.

- `ack`
  - Id – The identification number returned by the server to represent the device in future communication.

- `get_cap` – This originates from the server when a device with a `src` device type registers. The server will send this message to retrieve the content on that device.

- `request`
  - Capability – String representing the capability to be queried, typically `content_list`. See Table 4.1.

  - `ack`
    - Capability – String representing the capability to be manipulated.
    - Value(s) – String, integer, list (of strings and/or integers) to set.

- `get_contents` – This message command is used by the remote control device to retrieve the global content list from the server.
  - `request`
    - No Arguments
  - `ack`
    - List – List of strings representing the names of the content filenames.

- `notify` – A general mechanism for a a client or server to send an out-of-band message to another device.
  - `request`
    - Id – Notification identification number; in this case it's used to notify the remote control that new content or devices exist in the network and that it may pull this information from the server.
  - `ack`
    - No Arguments

- `get_sinks` – This command is used by a client to retrieve the global list of sinks in the network from the server. Typically this command is sent by the remote control.
  - `request`
    - No Arguments
  - `ack`
    - List – List of strings representing the sink names (which are given during registration).

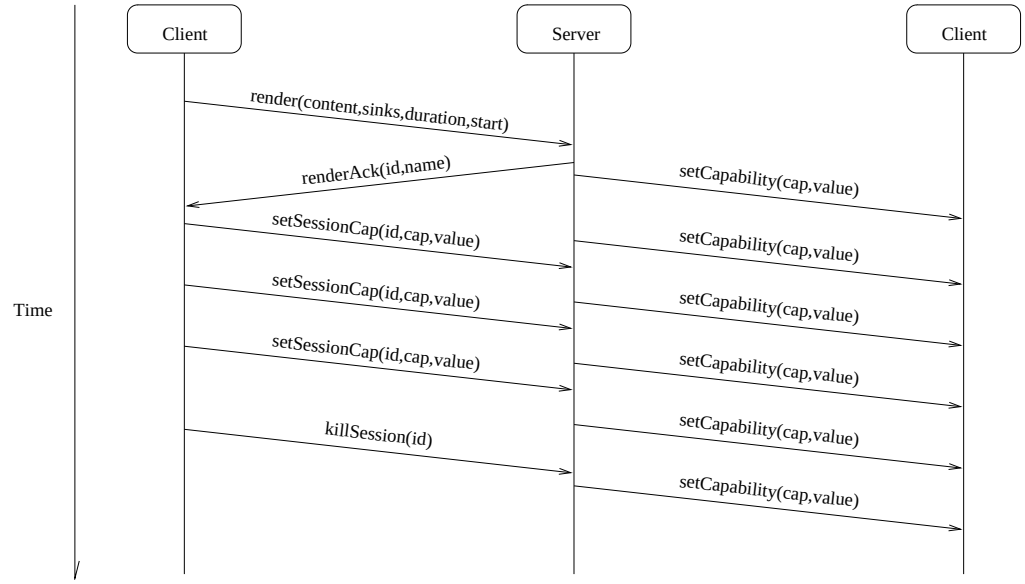


Figure 4.6: DCP session creation using the `render` command, followed by a set of messages to manipulate the session.

4.2.3.2 Session

A session is created by an application (*e.g.* remote control) when it wants to stream multimedia content from one device (source) to one or more other devices (sinks). A session is created by the application via the server. Sessions encapsulate a set of devices (*i.e.* sources, sinks, possibly transcoders), routing between these devices, and state (*e.g.* current position in playlist). The DCP contains commands to create a session, manipulate a session, and destroy a session. Session creation is accomplished with the `render` command. This command contains information about what content to render (a playlist) on which device(s), the duration between elements of the playlist, and starting position in the playlist. After session creation, the server will reply with a session identification number which is used in subsequent manipulation or destruction of the session. Session manipulation can take the form of setting a session capability which the session inherits for the involved devices. Figure 4.6 illustrates this process.

The following command messages may be used for the creation, manipulation, and destruction of a session. For each message command and message type, a description details its general usage and arguments follow. Again, for brevity, `error` message types are not included.

- `render` – Used by applications to create a session.
  - `request`
    - Content List – Contains one or more string representing content file names.
    - Sinks List – Contains one or more strings representing names of sink devices.
    - Duration – Duration for each element of the content list. This can be though of as the time to display a photo in a slide show or how long to preview a song in the playlist. If this is set to 0, the content element is assumed to have its own duration.
    - Start Position – The content element to begin playing.
  - `ack`
    - Id – The identification number assigned by the server to be used in subsequent manipulation or destruction of the session.
    - Name – The name of the session.
    - Capabilities – A list of capabilities of the session. For example, a multimedia session has the ability to play or stop media, select the next or previous content in the playlist, etc. See Table 4.1.
- `set_session_cap` – This command is used to set a capability of a session. Typically, this command is issued by a remote control to the server.
  - `request`
    - Session Id – Identification number assigned by the server to represent the session.
    - Capability – String representing the capability to be manipulated.
    - Value(s) – String, integer, list (of strings and/or integers) to set.
  - `ack`
    - No Arguments.
- `set_cap` – This command is used to set a capability of a device; this different from `set_session_cap` in that it is issued by the server to a device, rather that from a remote control.

- `request`
  - Capability – String representing the capability to be manipulated.
  - Value(s) – String, integer, list (of strings and/or integers) to set.
- `ack`
  - No Arguments.

- `kill_sessions` – This command destroys the session state in the network and resets the devices participating in the session. This command is typically issued by the remote control to the server.
  - `request`
    - List – A list of session identification numbers.
  - `ack`
    - No Arguments.

4.2.4 Library Interface

The DCP described in this section is provided to the client programmer in the form of a library interface. The library core contains definitions for the packet formats (commands, types, message bodies) and helper functions for packet sanity checking, parsing, and construction. This functionality is wrapped in a client library which handles argument processing (command line), threading, command handlers, server connections, and logging.

Argument processing is provided to ease the implementation of client programs. When clients are started, they must know the server address and port, logger address, and their own location. This part of the library provides a standardization for processing arguments for all clients, as well as, a standard format for providing argument via the command line.

Threading support is important to get correct because typically clients are multi-threaded. One thread will be handling the incoming DCP messages and another thread will be performing some client processing (*e.g.* graphical user interface). Furthermore, multiple threads may be manipulating data structures. The library provides a standard way to create and start a DCP interface processing thread and provide it a lock which it must use when processing all DCP commands (*i.e.* which may alter shared variables). If there is no worry of race conditions on shared variables, a null lock handle may be provided.

The library provides a common framework to handle messages. All messages are supplied with a default *nop* command handler that essentially prints "Not Implemented" to stdout. Users of the DCP library can override these default handlers with their own client-specific handlers. The client library defines a registration mechanism for handlers and when the client receives the command message, the message is dispatched to the user specified handler.

Establishment and maintenance of connections from the client to the server are handled by the DCP library. Initially, the client prepares a data structure which contains the necessary fields to maintain the connection. This data structure contains the server address, port, and a new connection thread. The library uses the thread to establish the connection. If for some reason the connection is not able to be established or fails, the client library will attempt periodic retries until the connection is re-established. When the connection is established or dropped, special user-registered methods are called by the library. These methods are useful for cleaning up or initializing state, or indicating to a user of a GUI that the connection to the server has undergone a change.

Finally, the DCP library standardizes the logging streams that a client can produce. Log message are tagged with a client-defined preamble. Log messages are binned into streams according to pre-specified types. A client can specify log messages types in their code along with the actual log message. These log messages can be configured by the client developer to be transmitted to stdout, to a file, or over the network to a centralized logging listener. All log messages can be filtered based on the log's type.

4.3 Clients

Client devices are those which have a capability or capabilities to offer to applications in the Home Gateway Demo. Clients are distinguished by a unique set of capabilities which are discovered through the registration. Each client specifies in the registration the capabilities it can perform and the device type(s) is supports (see Tables 4.1 and 4.2). Currently we have four categories of clients: remote control, content renderers (audio and video), content sources, and location sensors. All clients and the server are networked with IP over 802.11g and Ethernet. IP addresses are assigned statically.

This section will discuss the purpose of each client, briefly how each functions, and some of their implementation details. For more information on how to download and compile the source for the demo, and operate the clients and server, see Appendix B.

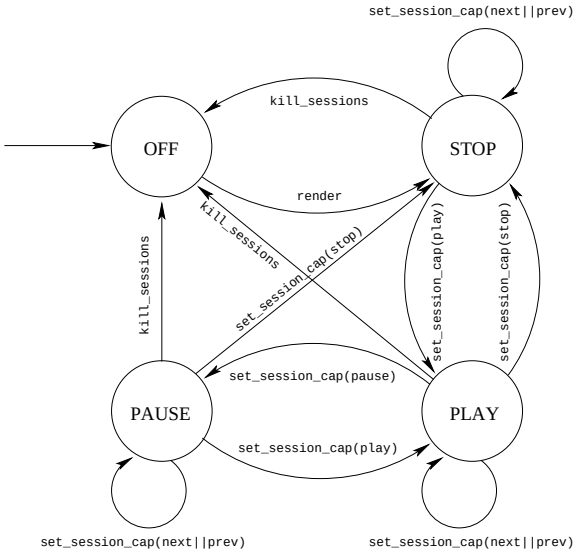


Figure 4.7: Remote control state machine.

4.3.1 Remote Control

The purpose of the remote control is to run an application which allows the user to create sessions. This is accomplished by the remote control registering with the server, retrieving a content and list of sinks, and displaying this to the user. After registration the user can create a playlist of content and a list of sinks, and initiate a session using the `render` command. After initiating a session, the session can be manipulated by choosing play, stop, next, previous, etc. This is illustrated with a state machine in Figure 4.7. Currently, we have two versions of the remote control, one which provides a simple command line interface and another which runs on the Nokia 770 handheld Internet tablet [20] (see Figure 4.8).

The remote control is written in C using the client library described in Section 4.2.4. The GUI version is written with Maemo GTK++ widgets [17] and is operated with a stylus. The GUI is illustrated in Figure 4.9. Additional data structures are used to managed the content, sink, and session data pulled from the server. In the rare case of server or connection failure, the remote control cleans up this state and attempts to reconnect to the server.

4.3.2 Content Sources

The purpose of the content source is to push a stream of content to a destination device. Initially all content sources register their capabilities with the server. Then they listen and respond to content queries from the server. When the user creates a session via the remote

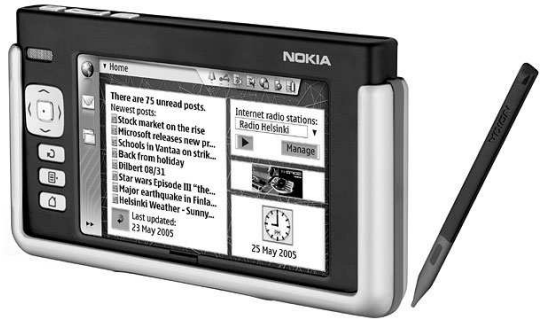
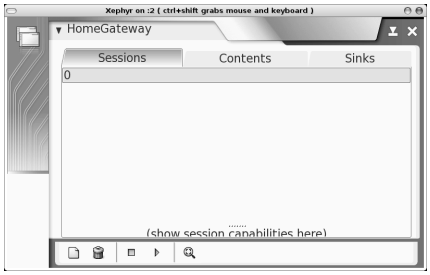
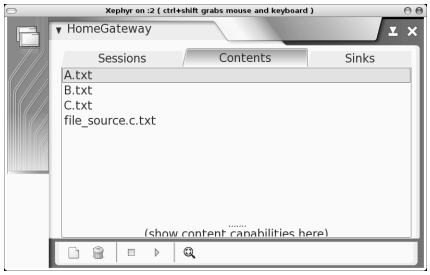


Figure 4.8: Nokia 770 handheld Internet tablet.

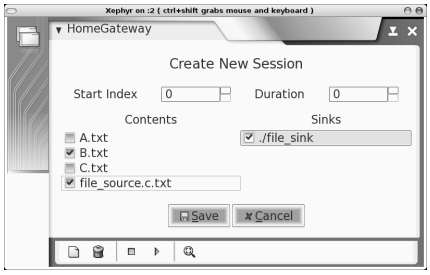


(a) Sessions tab.



(b) Contents tab.

(c) Sinks tab.



(d) Creating a new session.

Figure 4.9: Remote control application for Home Gateway Demo.

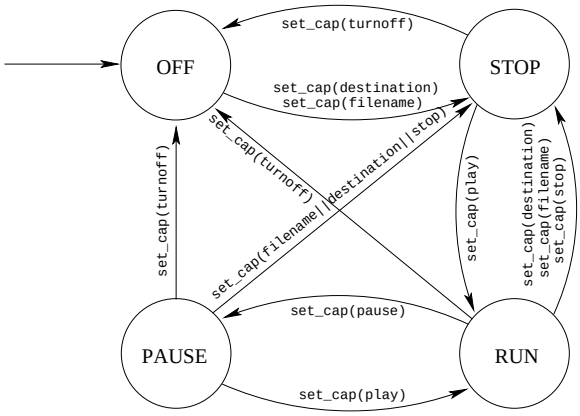


Figure 4.10: Content source state machine (push semantics).

control, the content source is instructed to stream a particular file (with `filename` capability) to a particular destination (with `destination` capability). When the user instructs the session to play, pause, stop, next, or previous, these are translated into the corresponding `set_cap` commands and set via the server to the content source. The operation of a content source is illustrated in Figure 4.10.

Currently the content sources are implemented on two Apple Mac Minis running Debian Linux. One content source provides video, while the other provides audio and still photographs for a slide-show. Two sources were used to model a distributed content repository. The content source client is implemented in C, using the client library described in Section 4.2.4. Buffering for audio to avoid poor audio quality (*e.g.* "clicks" or "pops") is accomplished using VideolAN Client (VLC) [26]. While it would be desirable to stream video using VLC for similar reasons, currently we are not able to do this. This is because our hardware implementations of video decoders and encoders do not conform to complete video standard specifications that VLC requires. Instead, with reasonable quality, video is streamed is over a TCP socket which provides buffering, but no rate control.

4.3.3 Content Renderers

The purpose of content renderers is to receive a pushed stream from content sources. Like all other devices, content renderers register their capabilities with the server, most importantly their port signatures. These port signatures are used to determine what type of stream can be accepted and whether or not transcoding is required. After registration, the content renderer awaits command by the server to prepare for a specific content type: the `set_cap`

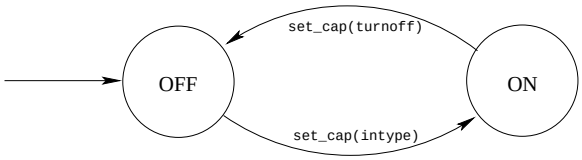


Figure 4.11: Content renderer (sink) state machine (push semantics).

Figure 4.12: Xilinx XUP development board used for content render clients.

command with capability `intype`. Next, the content renderer awaits the stream on the port specified (in its port signature) for the specific media type. Content renderers can be configured to accept multiple types of content, thus the need for the port signatures and server specifying which port will be used. Figure 4.11 shows the very simple state machine which captures this operation.

Currently, the content renderers are implemented as C programs (using the library described in Section 4.2.4) executing in Linux. Linux is running on the embedded Power-PC processor of the Xilinx XUP development boards shown in Figure 4.12. A base system which includes the necessary gate-ware for networking, buses, memory interfaces, video frame buffer, as well as, audio drivers and file system have been built for the XUP board. We decided to use this platform because we require a high degree of programmability at both the software and hardware levels. At the software levels, we are running our DCP clients; at the hardware level, we can use custom built MPEG-4 (from Xilinx) and Motion-JPEG video decoders to perform video decoding in real-time. Current consumer off-the-shelf (COTS) devices would not be able to support these features.

Figure 4.13: Mica2 sensors used for location sensor client.

4.3.4 Location Sensors

Location sensors are responsible for detecting the location of a user. The location is transmitted wirelessly to the server which can use this information to customize the experience for the user. For this demo, the location of the user is used to implement a scenario in which the video stream "follows" the user. Currently, the location sensors transmit which of three virtual rooms the user is located. Each virtual room contains a content renderer. The location sensor client aggregates information from all sensors and sends one event to the serer when the user changes rooms. Like other clients, the location sensor client registers with the server.

The location sensors are implemented with pressure mats attached to *mica2* [2] sensor network motes. These motes are shown in Figure 4.13. The pressure mats are attached to the mote's on-board A/D converters and a simple TinyOS [24, 14] program reads the output of the A/D converter to send a packet to the location sensor client. The location sensor client, implemented in Java, then converts this packet to a DCP `notify` packet to send to the server.

4.4 Server

The Home Gateway Demo is centrally managed by the server – the run-time system that manipulates devices, session state, content listing, and handles discovery. In addition to

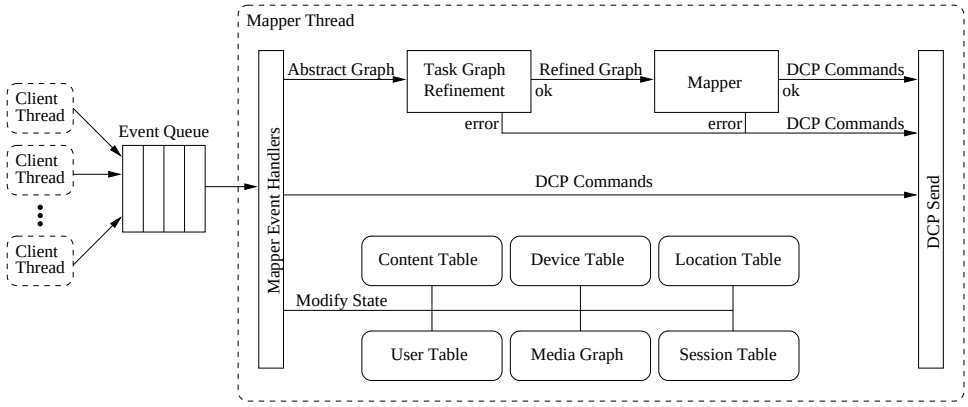


Figure 4.14: The server contains two types of threads: client threads and mapper thread.

these routine tasks, the run-time system contains a resource manager, which "implements" the sessions as requested by user via the remote control. This corresponds to the discussion in Section 3.4. The resource manager automatically deals with several types of constraints: mismatch in content type between the media source and sink, and change of user or device location. Whenever user makes a request through a remote control or changes his/her location, the resource manager re-evaluates specified tasks to meet all constraints.

4.4.1 Overview

Our goal was to the make the server implementation simple and manageable, allowing us to focus on control algorithm development, rather than performance optimization. The server consists of two types of threads (see Figure 4.14): *N* client service threads and one "mapper" thread. A client service thread is responsible for a single client, (*i.e.* a content source/sink, remote control, etc.). Client threads are created when the client first registers with the server. This thread will service all client initiated communication until the client either dies or is explicitly shut down. The service thread receives messages from its client and converts them into events, which are added to the mapper thread's event queue.

Events are dequeued and dispatched by the mapper thread to the appropriate event handler based on the event's DCP message command and message type. In many cases, the mapper events will trigger the server to reply to the original client directly. Alternatively, events may only modify the server state. If the message contains a `render` command, it is processed by a pipeline of Task Graph Refinement and Mapper elements (see Figure 4.14), which implement a subset of functionality of a resource manager described in Section 3.4. Task Graph Refinement takes an abstract description of the session as submitted by the

`render` command. Along with this command are some implied constraints, such as rendering should be local to the user and media data-format types must match. This part of the mapper thread produces a mapping from abstract graph to refined graph. The Mapper stage (see Figure 4.14) then converts the refined graph to a sequence of DCP commands to set up the communication between clients. Additionally, `set_session_cap` commands will also flow through this pipeline affecting the mapping of the session, effectively implementing run-time adaptation.

4.4.2 Data Structures

The mapper thread consists of a set of algorithms and data structures; the latter will be discussed here. Many of these data structures are relatively simple – a mapping between one object and another, however, there are two which are more complex: the session table and media graph. Most data structures provide methods which take or produce DCP commands directly, allowing most of the packet handling to be done internally to simplify the server code. To avoid having locks guard access to data structures, only the mapper thread is allowed to access them. Client threads can only affect the state of those data structures by enqueueing an event to the event queue and having the mapper thread modify the state. The following sections will give some brief detail about each of these data structures:

4.4.2.1 Content Table

The content table is a list of content objects. Each content object maintains the:

- content's string name,
- the host device,
- and the content's media type.

The media type is used to determine if transcoding is necessary (see Section 4.4.2.5). Content objects may be added or removed from the table, asked to return their host device identification number, and asked to infer their media type based on their content file name.

4.4.2.2 Device Table

The device table keeps a mapping between an identification number and a device object, as well as the device's name given during registration and the device object. Each device object contains:

- the device's connection handle (used to send and receive DCP messages),
- location,
- name,
- identification number,
- capability set,
- input port signatures,
- output port signatures,
- device types,
- and device handler (for sequencing DCP commands).

The device handler is analogous to the HAL described in Section 3.4.4.1. It is a specialized interface the server uses to communicate DCP commands to the actual device. It abstracts the construction and sending of DCP messages to a higher level interface corresponding to the state diagrams shown above.

The device table also contains a special mapping of bins for sink devices. When a new device registers, if it is a sink device, it will be placed into a bin (or bins) which represents the general nature of that sink. For instance, a sink device which has ports in its signature corresponding to a video standard would be placed in the `VIDEO` bin. Similarly, if it also had an MP3 port, it would be placed in the `AUDIO` bin. The importance of bins are evident when a remote control asks for a listing of sink devices. In addition to the absolute names, the server also returns bin names to the remote control. The remote control user can then create a session containing a bin name as a sink device. When that session is implemented (see Section 4.4.3), the bin name will be resolved to a device which is closest in proximity to the location of the user. This is how we implement the scenario in which the media stream "follows" the user.

4.4.2.3 Location Table

The location table maps a user or device to a valid location. Valid locations are read from a configuration file upon server startup. When registration occurs, the location of the device or user is checked against valid locations. If the location is not valid an error is returned. While running the demo, the user is typically the one who changes location the most. This

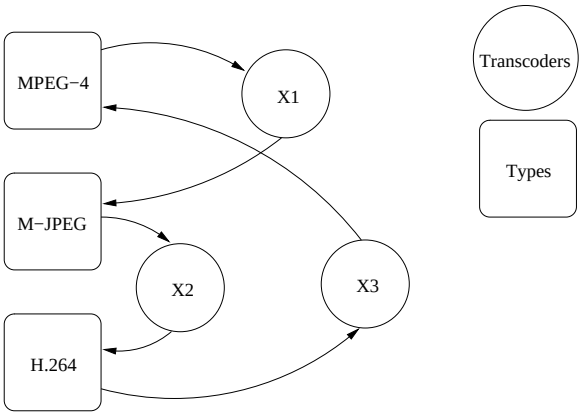


Figure 4.15: Media graph data structure is a bipartite graph consisting of media types and transcoder nodes.

occurs when `notify` DCP message are received from the location sensor client. The server parses these messages and updates the location table.

4.4.2.4 User Table

The user table is a simple map between a user name and a user identification number. The table keeps its own counter for user identification numbers. Users are registered when an application comes on-line. For now, since users, applications, and a control client (such as the remote control) are all the same, the client's name is used as the user name when the remote control client registers.

4.4.2.5 Media Graph

The media graph data structure is a bipartite graph $G = (V_1 + V_2, E)$ such that V_1 is all known media types, and V_2 is all known transcoders. The existence of an edge from a type node A to a transcoder node, then from the transcoder to a type B, means the transcoder node can convert type A to type B. Figure 4.15 shows an example media graph. A simple path may only consist of a single transcoder, such as the path $\{$MPEG-4, X1, M-JPEG$\}$. However, suppose the application required a conversion between MPEG-4 to H.264 media types. In this case a compound path is required: $\{$MPEG-4, X1, M-JPEG, X2, H.264$\}$.

The current usage of the media graph is very simple: we only have two kinds of types (MPEG-4 and Motion-JPEG) and one transcoder. New transcoder nodes are added when transcoders register; new type nodes are added if the input/output types of a transcoder

represent an unrecognized media type.

4.4.2.6 Session Table

The session table maintains a mapping between session identification number and a session object. Each session object contains the state of a session:

- user identification number (owner of session),
- session identification number (assigned by server when it receives a `render` command),
- session name,
- abstract graph (see Section 4.4.3.1),
- and refined graph (see Section 4.4.3.2),

The session table keeps its own private counter for assigning session identification numbers to incoming sessions.

4.4.3 Resource Manager

The job of the mapper thread is to process events from the clients. These events could be simple events which modify the state. For example, when a client registers, a new entry in the device table is inserted containing the registration information. Additional DCP messages may be generated and sent to the registering client. Similarly, a device may fail and server state may need to be cleaned up. However, some events may be more complex. An example of a complex event is one that deals with sessions. For example, session creation (`render` command). Similarly, an event which contains a message from the remote control client which requests the next song in a playlist be started, or the current video to stop. For these complex events we must invoke the first stage of the processing pipeline shown in Figure 4.14: Task Graph Refinement.

4.4.3.1 Task Graph Refinement

Task graph refinement takes as input an abstract description from the `render` command containing a playlist of content and a list of sinks (represented as a graph). This input, known as an *abstract graph*, is shown in Figure 4.16(a). Notice there are two types of nodes in this graph, a sequence operator and a set of sinks. Sequence operator nodes correspond

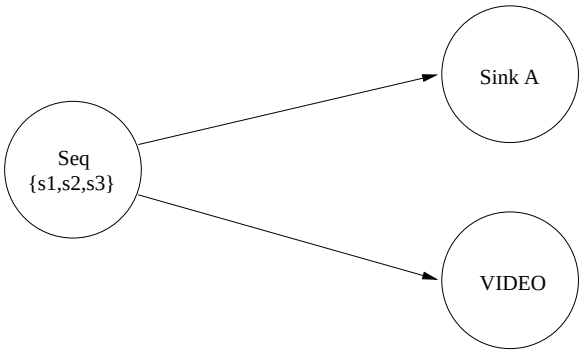
to the playlist of the `render` command. A set of sink nodes correspond to the sink names listed in the `render` command. Notice that one of the sink nodes in underspecified with the bin name VIDEO. This will be discussed further below. The output of the refinement process is another graph called a *refined graph*.

An example of the refined graph is shown in Figure 4.16(b). This graph contains nodes which represent in more detail the functionality the network can support. Some nodes in this graph are the same, others are more detailed, and still others are new. Source nodes are a refinement of the sequence operator node. Sink nodes in some cases may be exactly the same, others may be a more detailed (refined) version of a sink. For example, a sink node in the abstract graph may represent a bin device type. In this case, the refined version of the bin name, say dependent on the user's location, is an actual device. In this example, Sink D is a refinement of bin VIDEO given the user's current location. Finally, new nodes represent transcoders. The transcoder nodes are the set of modules that must process a media stream to match the media data formats between the content of a source and the input type of a sink. Transcoders are only inserted if the sink input port signature differs from the source content type.

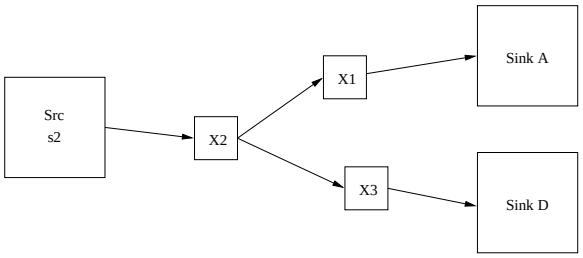
Refinement begins by resolving the sequence operator to the actual device by inspecting the state of the render command. Given the playlist and a start position, the content file name is used to query the content table and retrieve the host device. Next, the sink device is resolved. If the sink device is fully specified, then this is simple, however, if it is not, then user location is used to resolve the device. For example, if the sink device is of a bin type, the refinement processes will ask the device table to resolve the bin to a specific device given the user's location.

Next, any media data format mismatches must be resolved by inserting transcoder nodes into the refined graph. If the formats are the same, the source and sink(s) nodes are directly connected. Otherwise, the insertion of transcoders is accomplished by using the media graph data structure described in Section 4.4.2.5. The data format of the source and sink nodes must also be present as type nodes in the bipartite media graph. Given the resolved sink's input format and the content format of the source, a series of transcoders (or path of transcoders) is found which bridges the data formats. The algorithm applied to the media graph data structure is discussed in Section 4.4.3.1.1.

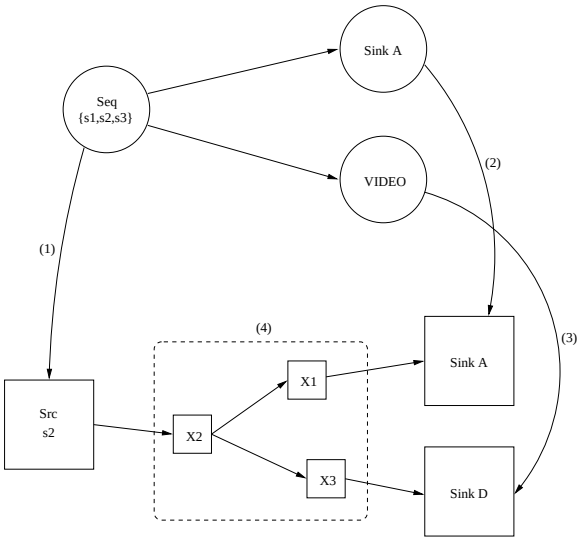
After the successful insertion of transcoding nodes, the refined graph is now ready to be mapped (in the language of Chapter 3, it is ready to be allocated). The mapping is the responsibility of the next stage in the pipeline described in Section 4.4.3.2. Algorithm 1 gives the pseudo-code for the entire refinement processes shown in Figure 4.16(c). Note



(a) Abstract graph containing playlist (sequence operator) and sinks from `render` command.



(b) Refined graph with source, transcoder, and sink nodes.



(c) Steps from abstract to refined graph.

Figure 4.16: Refinement process.

the numbers in parentheses for the Algorithm correspond the the steps in the Figure.

Algorithm 1 Algorithm to convert an abstract graph to refined graph.

```
1: source ← resolve(seq) (1)
2: insert(refined_graph, source) (1)
3: for all sink in sinks do
4:    if sink is abstract then
5:       sink ← resolve(sink) (3)
6:    end if
7:    insert(refined_graph, sink) (2,3)
8: end for
9: match_types(media_graph, refined_graph) (4)
10: return refined_graph
```

4.4.3.1.1 Media Type Matching Currently the algorithm for media type matching finds any path in the media graph (see Section 4.4.2.5) and inserts these nodes along this path as transcoding nodes in the refined graph. What complicates this algorithm is the ability to perform incremental graph construction to share as many nodes as possible from a sink to multiple sources. Incremental graph construction is used when the resource manager adapts to a change in the environment, such as the user moving to a different location. In this case, the existing graph is re-used as much as possible. Algorithm 2 illustrates how this is accomplished.

Algorithm 2 Algorithm to find media graph paths for a single source and multiple sinks.

```
1: source
2: graph
3: for all sink in sinks do
4:    path ← find_simple_path(source, sink)
5:    if found path then
6:       merge_path(graph, path)
7:    else
8:       cleanup and abort
9:    end if
10: end for
11: remove_unused_nodes(graph)
12: return graph
```

The merge path function essentially walks the existing graph attempting to find nodes of the newly formed path. As long as it finds these nodes, no new nodes are added to the

existing graph. Once the path diverges from an existing path in the graph, new nodes are added to accommodate the new sink. Finally, when all the paths have been successfully added, unused nodes from the original the graph must be removed. These nodes are identified easily because they are unmarked after walking the existing graph. Nodes which are part of new paths, or reused paths, are marked during the path merging step.

This algorithm becomes more complex when one applies constraints to the choice of paths. For example, if the application required a particular resolution, QoS, etc. for the pipeline of transcoders, then the path selection would not be as trivial. Furthermore, this algorithm would become more complicated if each transcoder is allowed to have an in-degree and out-degree greater than one[1]. Currently, both of these issues are left to future work.

4.4.3.2 Mapping

Mapping is the process of allocating the nodes of the refined graph to the available resources in the network. For now, allocation is first-come, first-serve and therefore arbitration is not an issue. Since source and sink nodes of the refined graph contain sufficient information, they are mapped directly to their corresponding physical devices. In this case, the mapping is simply one-to-one. Transcoder nodes are not associated with a particular device. In our implementation, compute nodes (*i.e.* FPGAs) exist for the purpose of implementing hardware transcoders. Each transcoder node is mapped to a compute node. Since there are four slave FPGAs on the BEE2 system (each is a compute node), they are reserved as needed for the mapping of transcoder nodes in the refined graph. For each transcoder, a configuration bit stream is loaded over the board interconnect via a driver that makes the slave FPGA appear as a Linux device. Media streams bound for, or coming from, a particular transcoder are transmitted via the on-board interconnect and drivers. The mapping process is shown in Figure 4.17.

Once all devices are known, the final step of mapping requires the routing between devices to be established and the generation of DCP messages to configure the clients. Since source and sink clients are already part of the IP network, routing for these device is simpling setting the `destination` capability for the source device. If transcoding nodes are required, the necessary routing must be setup in the network and on the BEE2 board such that content can flow from the source over the network, to the board and the slave FPGA(s), then back to the content sink. Figure 4.17 illustrates the output of the mapping step. Once

[1] It is not acceptable to represent the same transcoder with multiple nodes because it can only be used for one stream at a time.

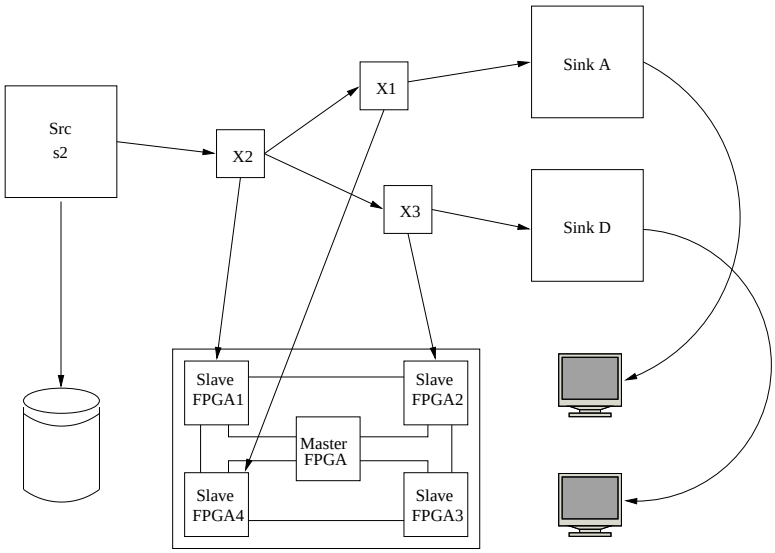


Figure 4.17: Mapping process.

the devices are known and routing is established, the abstract implementation task which we started with is now "implemented" in the network and ready to start. Appropriate DCP message are generated to setup the state of the clients and begin the streaming of content for the content sources.

4.4.3.3 Adaptation

Periodically the server receives events which pertain to a running session. Adaptation in the context of this demo is simply the application of refinement and mapping on existing abstract graphs (the session), given these events, over time. Adaptation of existing graphs is performed incrementally, meaning that some nodes of the existing graph may be reused, replaced, or re-tasked. Appropriate DCP messages will be generated to perform these jobs.

For example, suppose a new content rendering client comes online. This `register` command will insert the new object in the device table and initiate a re-evaluation of the abstract graph. When the sink node is refined it may map to the newly registered device. DCP messages will shut down the old source device, start the new source device, reset the source's destination route, and begin streaming the content.

Similarly, when the source reaches the end of a content file, a `notify` event will be sent to the server from the source. Again, the abstract graph's sequence operator is resolved to a new content file, which may require a new source device to be created to replace the old

source device.

Yet another example occurs when the location sensors send a `notify` event indicating the user has changed from one virtual room to another. The user's location is used to re-evaluate the location-dependent sink node. All of these events may change the result of the refinement process, which then requires re-mapping. Again, adaptation is simply the application of refinement and mapping in time.

While the resource manager creates an illusion of "continuous" remapping of sessions, in practice the system adapts only when the following events occur:

- `render` – the mapper adds the session to the Session Table. The session is refined and mapped. The session owner is notified if errors encountered.
- `kill_sessions` – the resource manager terminates the session and shuts down all devices allocated to it, removes the session entry, and notifies the session owner.
- `register` – a device came on-line. Session remapping may be required, in case the user requested to render a piece of content on the *closest* renderer to him, and the new device is the closest. The refinement simply checks whether this device affects any of the existing sessions and perform remapping if necessary.
- Device failures – a device disconnected. The affected sessions are remapped; those that *require* this device are shutdown, and the owners are notified.
- User Location Change – remap all sessions that are affected by the user by querying the Session Table.
- New Content – a new content source came on-line. The server will notify all remote controls.
- Content Unavailable – a content source disconnected. Affected sessions are remapped; session owners are notified.
- `notify` – a client sent the server a notification: "End of File" or "Location Change".
- `set_session_caps` – a remote control request to modify the session (*e.g.* play, stop, etc.). The resource manager may remap the session and then forward necessary commands to session's devices.

Chapter 5

Evaluation

This chapter will show an evaluation of a subset of the architecture implemented for the Home Gateway Demo. Many of the design decisions of the system architecture and subsequent implementation are difficult to measure quantitatively. This is because much of what drives the performance is based on qualitative properties of user's perception or the programmer's ease of programming. Furthermore, there are few established metrics or complete evaluations for other systems in the literature. Here we will attempt establish a set of metrics and demonstrate the results of our implementation given these metrics

5.1 Metrics

The experimentation will focus on the metrics of performance and scalability. Our main question to be answered is: Can the server perform common operations in real-time? This performance metric is important for scenarios of streaming video and changing the rendering device at run-time. Another question involves scalability. When there are many sessions, devices, content, etc., how does the server perform? Does it take more time to perform common operations when the number elements in the system is on the order of tens-of-thousands?

Performance will be measured as the time to execute common operations (*e.g.* `register` command) and common sequences of operations (*e.g.* creating and running a session). The time to execute these operations will be measured as CPU execution time by the server process.

We will also briefly discuss qualitative metrics. In designing the Home Gateway Demo, we had a set of design goals for streaming video, on-the-fly transcoding, location-aware

mapping, etc. We will attempt to answer how well we met these goals and what shortcomings exist.

5.2 Experiments

These experiments will focus on the performance of the server since it has implemented a subset of the architecture described in Chapter 3. In particular, we will focus on the performance of the server with respect to the processing of DCP commands and refinement/mapping of tasks graphs. Again, this performance measurement relates to the user experience (*e.g.* how much lag is there to start a session?).

5.2.1 Setup

The experiments were performed using a Lenovo Thinkpad T60. The specifications of this machine are in Table 5.1[1]. A single machine was used instead of multiple machines because it could be controlled more closely and these experiments factor out the network, isolating the performance of the server process. There are two processes communicating through the loop-back interface on the laptop. The two processes are the test driver and the server. The test driver initiates a connection to the server and special out-of-band DCP messages are used to run experiments. The test driver impersonates actual devices. Impersonation is desirable because we don't actually stream data, nor do we care about the performance of the client devices. Instead, a virtual network of artificial devices is created and the server process doesn't know the difference. The server process is running in a special mode which removes all debug printing and sending of messages to clients (because they do not actually exist). During the running of an experiment, DCP messages are created in the server and dispatched directly to the mapping thread, bypassing the event queue. Figure 5.1 illustrates a generalized picture of the setup.

All measurements are CPU execution time for the server process. This was done to achieve the most accuracy possible and to exclude other processes running on the system. CPU execution time was measured with the `clock()` syscall. A start time was captured before the experiment began and an end time was captured when the experiment finished. The difference between end and start times, divided by the number of iterations in the experiment, resulted in the average CPU execution time. In order to make an accurate measurement a minimum of $10,000$ iterations were performed for each experiment. For

[1] Frequency scaling was turned off.

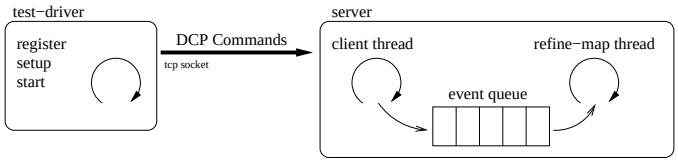


Figure 5.1: Generalized setup of experiments.

Table 5.1: Test machine hardware specifications.

Component	Properties
CPU	Intel Core Duo, 1.83Ghz
Memory	1GB DDR2 PC2-5300
Disk	80GB SATA
Network	Gigabit Ethernet; 802.11a/b/g

most experiments, this translates into processing a single DCP message for each iteration. The general format of each experiment looks something like the following:

```
start = clock();
for 1...iterations {
... // process DCP command(s)
}
end = clock();
average = (end-start)/iterations
```

For each of the experiments, a brief description will be given:

5.2.2 Registration

The initial set of experiments were performed to measure the time to process register commands for different types of clients. This was done by creating and issuing register commands for sources, transcoders, remote controls, and sinks. Each of these commands were processed by the server in independent runs, meaning the server was restored to its initial state before each registration experiment.

5.2.3 Mapping

Next, different types of DCP commands which trigger the execution of the resource manager (*i.e.* task graph refinement and mapping) were measured.

The first re-mapping scenario included two sources and one sink. The sequence operator of the abstract graph contained two content files. Each source would be alternately mapped based on the index of this sequence operator. This re-mapping was triggered by executing a `set_session_cap` DCP command with the `next` capability. An example is illustrated in Figure 5.2(a).

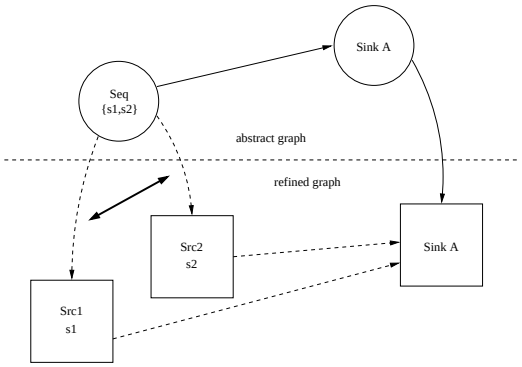
The next scenario was a re-mapping of sink devices. The scenario is setup with one source, and two sinks, each at different locations. The sink devices were specified by a generic location-dependent bin name in the abstract graph. The server processes alternating `notify` DCP commands with for a location change, each time re-mapping the sink a different location. Figure 5.2(b) shows this scenario.

Re-mapping between transcoding devices required a similar setup to the re-mapping of sink devices, however, the input types of the sink devices were different. When the `notify` command was received, the type mismatch would be detected and the appropriate transcoder would be inserted as illustrated in Figure 5.2(c).

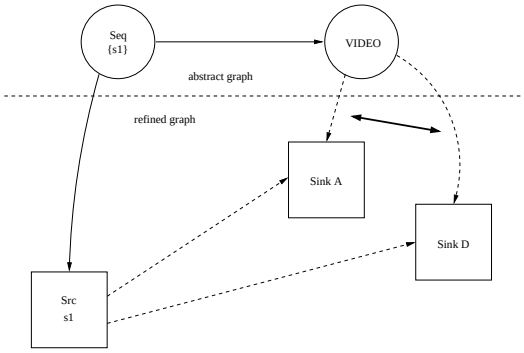
The final mapping experiment was performed by issuing a `render` command, followed by a `kill_sessions` command. This experiment will show the performance of the initial mapping of an abstract graph to a refined graph.

5.2.4 Session Manipulation

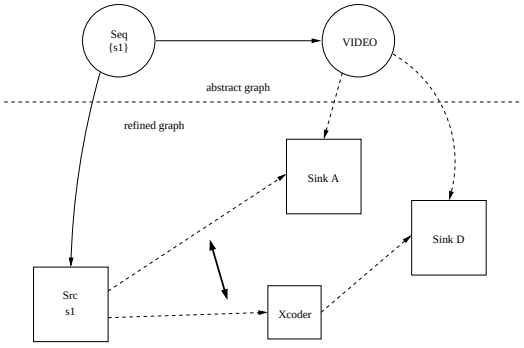
The final experiment was to measure the time to processes sequences of commands typically used during a session under different scales of devices, content, and sessions. The setup includes the registration of N sinks, N sources, and N content, where N ranged from $2,000$ to $40,000$. Once the setup was complete, the experiment begins. The experiment starts with the creation of N sessions using the `render` command. Next, each of the N sessions is manipulated by four different `set_session_cap` DCP commands (*e.g.* `play`, `next`, `previous`, `stop`). Finally, the N sessions are destroyed using the `kill_sessions` DCP command. Note that for this experiment, there is no re-mapping to different source or sinks device, except for the initial mapping upon session creation. Similar to other experiments, the average time is measured by taking the difference between the start and end and dividing by the number of iterations N.



(a) Source re-mapping



(b) Sinks re-mapping



(c) Transcoder re-mapping

Figure 5.2: Re-mapping scenarios.

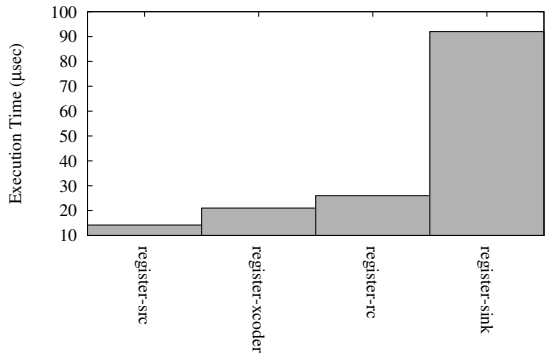
5.3 Results

5.3.1 Registration

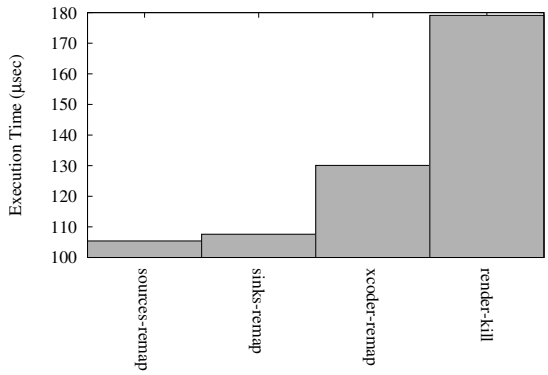
The execution times measured for registration of clients were as expected. The registration of a source, transcoder, and remote control required less than 30μsec each, while the registration of a sink was approximately 90μsec. This is illustrated in Figure 5.3(a). The reason why a sink takes more time to register (register-sink) than the other clients is it must be binned in the device table. This binning process must parse the newly registered device's types and compare each to the existing bin types. Depending on which bins exist, the device will possibly create new bins or be append to existing bins. The registration of a remote control (register-rc) takes slightly longer than other registrations because when a remote control registers, several notification messages must be manufactured and sent back to the remote control. These messages notify the new remote control of existing sessions, sinks, and content. Transcoder registration (register-xcoder) requires the next largest execution time because it must be added to the media graph. Finally, the source registration (register-src) takes the least time because it is a fast operation to append the source device object to the device table without generating other DCP messages or touching other data structures.

5.3.2 Mapping

The mapping results were also as expected. The results are shown in Figure 5.3(b). By far, the most time consuming operation is the initial `render` command (render-kill). This operation takes approximately 180μsec. While this experiment coupled this command with the `kill_sessions` command, the time is dominated by the rendering of the session rather than its destruction (see Figure 5.4(c)). Rendering a session requires the refinement and mapping algorithms to touch several key data structures: content table, device table, and the media graph. While we've tried to make operations on these data structures efficient, processing this DCP command requires a lot of heavy lifting. First, it must create the abstract graph and then refine that graph via the aforementioned data structures. Finally, the algorithm must acquire handlers for each device, manufacture appropriate DCP commands, and send them to the client devices for session initiation. Subsequent re-mappings typically only modify small portions of these graphs and usually do this in-place. Transcoder re-mapping is the second-most time consuming operation at approximately 130μsec. Again, this operation requires the creation and destruction of refined graph nodes, additional device handler acquisition, and manufacturing of DCP commands. Furthermore, the choice of



(a) Execution time of `register` commands.



(b) Execution time for server to process commands resulting in mapping.

Figure 5.3: Execution time for various DCP messages and sequences.

which transcoder to instantiate must be established by operating on the media graph data structure and re-building the refined graph incrementally. Finally, the sources and sinks re-mapping (sources-remap and sinks-remap, respectively) take approximately the same amount of time (105 to 110μsec). These re-mappings are faster to process because they only require re-resolving a single node of the abstract graph. This node is modified in-place in the refined graph.

5.3.3 Session Manipulation

This experiment demonstrates the performance of common session manipulation operations under varying table sizes. Here, table sizes means that every table, such as session, content,

device tables, were on the order of N, where N varies along the x-axis in Figure 5.4. Figure 5.4(a) shows the results of the initial session manipulation experiment. Most operations are virtually unaffected by the size of the table and take less that $100\mu sec$. Three DCP commands clearly depend on the size of the tables: `render` and `set_session_cap` with `next` and `previous` capabilities. Using a profiler, we found that these operations were iterating through lists when manipulating the content and device tables. By duplicating state and replacing the lists with hash tables, a simple time-space trade-off was used to dramatically reduce the execution time of these operations. The result of this optimization is shown in Figure 5.4(b) using the same scale, and using an zoomed-in view in Figure 5.4(c).

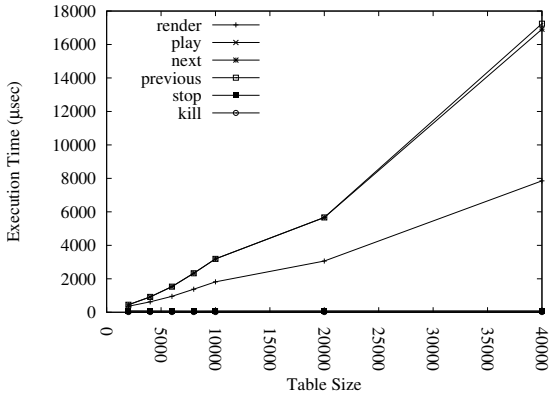
In Figure 5.4(c) one can see that the render still dominates the other operations, however, it is no longer dependent on the table sizes and reflects the results found in Figure 5.3(b). Operations using the `set_session_cap` DCP command to manipulate the session all have approximately the same performance and again reflect the results in Figure 5.3(b), however, in this case there is no re-mapping so these operation take a slightly less time of approximately $80\mu sec$. Finally, the `kill_sessions` command is the fastest operation at around $10\mu sec$.

5.4 Discussion

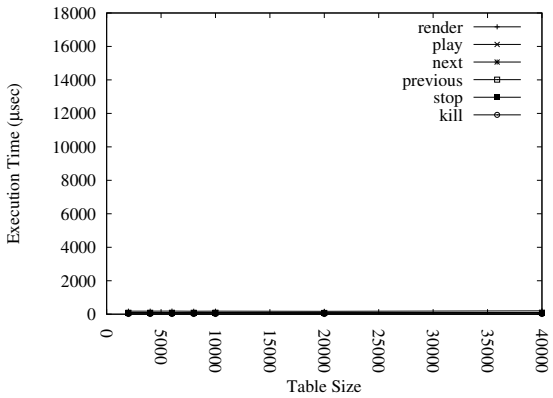
These results show the relative performance of common code paths through the server implementation which handle specific types of DCP messages. Under these conditions and assumptions, the server has relatively good performance, especially since our original design goals were for functionality and debugging ability, rather than performance. Theoretically it could sustain approximately $7,600$ transcoder re-mappings per second[2] before becoming saturated. This rate of processing should be adequate for current and future AIEs. However, one must also consider that these results are for simple session scenarios (single transcoder and switching between two sources/sinks). More complex interaction may require longer transcoder paths for larger sessions and this rate will likely be lower. Furthermore, these results are the raw performance of the server implementation and do not account for the performance of the network or the overhead of passing the event between the receiver threads to the mapping thread.

Another important property of the server implementation these results show is that the server is capable of efficient processing via the current data structures when the number of sessions, content, and devices scale to large numbers. In other words: the server appears

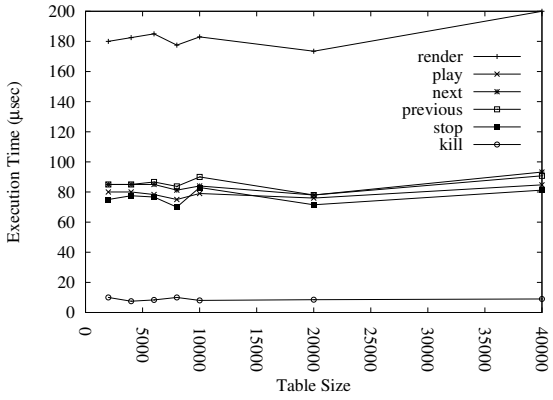
[2]This is difficult to measure in practice since the `clock()` interface only has a resolution of $10ms$.



(a) Execution time for various session DCP commands



(b) Execution time for various session DCP commands with optimization.



(c) Scaled version of Figure 5.4(b)

Figure 5.4: Execution times for session manipulation experiments.

to scale well. These results are modulo the complexity of the current implementation and the number of features we support. With more complex features, such as arbitrary abstract graphs or refinement/mapping constraints, the performance of the current data structures will have to be further tuned.

With regard to the user experience and ease of programming design goals, we feel that overall the implementation was successful at capturing an important subset of the architecture as well as meeting several goals left over from the first incarnation of the Home Gateway Demo.

On the user experience side, we wanted to support video transcoding in real-time. Using the MPEG-4 decoder and the decoder/encoder for Motion-JPEG, we were able to make a huge improvement in this area over the last demo. However, these hardware decoders where very time consuming to develop and they're still somewhat fragile. Our goal to include refinement/mapping was successful, but only for a very limited set of constraints: type mis-matching and user location. This ability was non-existent in the previous demo. Expanding the set of constraints and the type matching would be a worthy next effort. Finally, the ability to specify applications as abstract graphs was an important export of the research discussed in Chapter 3 as well and critical to making refinement/mapping successful. However, it is very limited in its expressibilty – only a single source and set of sinks. The ability to arbitrarily compose these graphs is a desirable property.

The ease of programming and integration of the demo was another important design goal. Client programming was an onerous task because each client was written independently and there were multiple code bases. To make the Home Gateway a successful design driver, we needed to make it easier for other research groups to "plug-in" their results. This time we merged code bases and factored out common functionality of the clients (communication with server), making it easier to develop a client. This client library was debugged well and had useful APIs. Another important lesson we learned was that we needed to improve observability. We added a logging infrastructure to the client library that all clients would use. These log messages could be sent to a file, standard-out, or over the network to a network-wide listener. Furthermore, the logs could be filtered based on their type. Finally, we wanted to reduce the time and complexity of setting up and running the demo. This was a secondary design goal and was aided by the common client library. Late in the development, we added scripts to automate the starting of clients.

Table 5.2 gives a report card summarizing the evaluation above. Overall our progress was a B+, meaning we have shown great potential, but there still remains much work to be done.

Table 5.2: Report card for qualitative design goals of user experience and programming.

Design Goal	Grade	Notes
Video transcoding	A-	Video was a huge improvement.
Refinement/mapping	A-	Still limited to location.
Rendering ability	B-	Abstract graphs still very limited.
Client programming	A	Factored out common functionality.
Logging infrastructure	A-	Improved observability.
Ease of demo setup and running	B	Need to automate.
Overall	B+	

Chapter 6

Opportunities for Future Research

One of the main contributions of this report is to outline a set of concepts that we believe will help realize an Ambient Intelligent Environment (AIE). While we have touched on many ideas, many of them have not been fully fleshed out. Those ideas which have been more fully developed, may not have been completely implemented. Opportunities for future research stem from these shortcomings.

One of the first steps to realizing the AIE is to clearly establish boundaries of this research. Currently, this work is attempting to address too many aspects of an AIE. Instead, we should focus on 2-3 application scenarios or usage models. For example, we may choose to focus on the multi-media aspects of these environments: ad-hoc collaboration between devices to share and stream multi-media content. Another example is sensor augmented usage scenarios. For example, customization of media rendering based on user location, preferences, etc. Building a completely general framework to solve all the problems is beyond the scope of this research project. By nailing down the boundaries of this research, we can successfully define where we will make an impact.

The next step is to develop a better application programming model and framework. How does a person write an application for this type of environment? The expected structure of a program beyond the vague description of a "network of tasks" should be refined. One solution may be to adopt a Model-View-Controller (MVC) structure. Currently, applications are tied to a control interface such as a keyboard, and a rendering device such as a video screen. MVC allows applications to be decoupled from these devices to adapt to new interfaces and devices.

In concert with defining the application programming model, we need to define a minimal middleware operating system to support the applications. In doing so, we will discover

what is different between a traditional operating system and the one we propose for AIEs. One difference is the fact that devices come and go frequently due to their mobility and wireless connectivity. What does it mean to have an operating system managing these ephemeral resources and what services are relevant to them? What abstractions are really necessary? We need to further develop the set of primitives which are useful for applications in this space. For example, a set of services such as content discovery, multi-media streaming, and device binding, eventing, etc.

Currently, we have a good start with our definition of the system architecture. We have established that applications must be abstract and mapped to the current execution platform. Furthermore, we have identified necessary services such as locationing and discovery. Future research will refine these concepts further. It would be a reasonable goal to eventually build a third version of the demo which incorporates these more developed concepts.

Once we have a functioning programming model and middleware OS, we must build applications which demonstrate this research. Two classes of applications come to mind: collaborative, ad-hoc applications which users share information and don't necessarily require infrastructure. For example, sharing of photos or video with friends in a public space. These applications must work successfully in spite of the lacking infrastructure (networking, storage, processing power, etc.). Another application space are resident applications which are long lived, such as home theater. These applications may be running constantly and recalled by the home users to perform a function. These applications will be highly dependent on the infrastructure.

Finally, we must develop better metrics for evaluation. Currently, there are very few established metrics for comparison of different AIEs system architectures. There are qualitative metrics which evaluate the "user experience." These are usually implementation dependent and difficult to compare directly. In addition to these qualitative metrics, quantitative metrics for performance, scalability, reliability are required. These metrics, along with established usages scenarios, can be used to place similar systems on equal footing, yielding clear advantages and disadvantages of each.

Chapter 7

Conclusions

Economic and technological factors have brought sophisticated electronics within reach of average users. These advances have driven the growth for many years in the personal computer space; they now drive the growth in consumer electronics for home entertainment, communication, automation, security, and monitoring. In the near future, wireless sensor networks will augment this nearly ubiquitous deployment of technology with ambient awareness. We define an Ambient Intelligent Environment (AIE) as an environment in which these technologies converge: a network of distributed devices seamlessly cooperating to simplify our daily tasks.

This is a very large research area which covers many disciplines. We have to chosen several of these research challenges where we feel we can make an impact. Specifically, we have made the following contributions toward realizing an Ambient Intelligent Environment:

- A set of abstractions, specifically for those elements affecting people, content, and system capabilities.
- A system architecture to enable configuration and organization of content and networked heterogeneous devices in a smart-home environment.
- A working prototype demonstrating several of the system architecture concepts.
- Performance results of this prototype.

In more detail, this report covered the following:

In Chapter 3 we discussed the main abstraction – the environment – and how it is broken down into personae, capabilities, and content abstractions. We discussed how we envision

the applications to be structured and execute, and how the system performs run-time refinement and mapping of these applications.

In Chapter 4, we detailed portions of the system architecture which were implemented in our prototype: the Home Gateway Demo. This included several clients for sourcing and sinking content, the remote control, hardware transcoders, and the server which orchestrates discovery, routing, and streaming for multi-media consumption.

Chapter 5 evaluated a portion of our prototype and gave results for client registration, session re-mapping performance, and demonstrated the ability of the server to scale. Through this evaluation, we were able to find and eliminate a bottleneck thus improving the server's scaling ability. Furthermore, we gave a qualitative evaluation of our design goals for user experience, as well as ease of programming and use of the demo. Based on this evaluation, we find that overall the Home Gateway Demo was a success but is still ripe with opportunity for improvement.

Finally, opportunities for future research were given in Chapter 6. Most of the future work consists of extending the set of concepts we have developed as part of our system architecture. In addition, future work on the Home Gateway Demo includes expanding the set of constraints used for mapping, the expressiveness of abstract graphs, and increasing the number of features, such synchronized audio/video.

Acknowledgments

As with most research projects, no one person can claim or accept all the credit. The success of the two projects discussed in this report – research on the system architecture and the Home Gateway – was the result of several people's efforts. I would like to thank my research adviser Jan Rabaey for his humor and generosity over the past year and a half, guidance and support for my current research, and helping me mold my future research. I would also like to thank professors Adam Wolisz, John Wawrzynek, and Jan Newmarch. Adam was instrumental in defining the set of concepts we use in the system architecture. His consistency and passion for detail could always be counted on during our discussions. John is the leader of the ongoing Home Gateway project (design driver for the GSRC). He helped to define the Home Gateway demo architecture and push us towards successfully implementing the advanced features such as video transcoding. Jan was a visiting professor who offered important insight and feedback on the system architecture, and how it related to the field of ubiquitous computing.

This project would have not been possible without the hard work and dedication of several talented graduate students. First and foremost, Yury Markovsky. Without his systems building insight (both hardware and software), this project would not have been successful. Furthermore, Yury is an important part of the ongoing system architecture research. I would like to acknowledge Stanley Chen and Alex Krosnov for their MPEG-4 and Motion-JPEG decoders, respectively. Stanley had the unfortunate, but ultimately successful, task of debugging an MPEG-4 decoder from Xilinx. Alex wrote and tested his decoder/encoder from scratch. He also helped with hardware integration. I would like to thank Kaushik R. for his work on VLC audio client early in the project, as well as Jana van Gruenen for her recent work to add location sensor support. Jana has also made major contributions to ongoing system architecture research. Finally, Bobak M. and Donald S., our undergrads who have been working to build hardware modules, remote control GUIs, audio-video synchronization, and debugging audio drivers.

Last, but certainly not least, I would like to thank my wife Lesley for putting up with me for the last six years. And of course, I must thank my parents, who have been models for work ethic and determination, both characteristics which have helped me be successful in college and graduate school.

This research has been funded by the Gigascale Systems Research Center (GSRC) as part of the Design Driver project. More information is available at `http://www.gsrc.org`.

Bibliography

[1] M. Coen, B. Phillips, N. Warshawsky, L. Weisman, S. Peters, and P. Finin. Meeting the computational needs of intelligent environments: The metaglue system. In *Proceedings of MANSE'99*, Dublin, Ireland, 1999.

[2] Crossbow Technology Inc. Mica2 Product Series. http://www.xbow.com/Products/Product_pdf_files/Wireless_pdf/MICA2_Datasheet.pdf, 2006.

[3] Digital Living Network Alliance. DLNA overview and vision white-paper 2006. http://www.dlna.org/, 2006.

[4] P.-Y. Droz. Physical design and implementation of bee2: A high end reconfigurable computer. Master's thesis, 2005.

[5] W. K. Edwards, M. W. Newman, and J. Z. Sedivy. The case for recombinant computing. Technical Report CSL-01-1, Xerox Palo Alto Research Center, April 2001.

[6] W. K. Edwards, M. W. Newman, J. Z. Sedivy, and S. Izadi. Challenge: recombinant computing and the speakeasy approach. In *MobiCom '02: Proceedings of the 8th annual international conference on Mobile computing and networking*, pages 279–286, New York, NY, USA, 2002. ACM Press.

[7] X. Fu, W. Shi, and V. Karamcheti. Automatic deployment of transcoding components for ubiquitous. Technical Report TR2001-814, New York University, Computer Science Department, March 2001.

[8] K. Gajos. Rascal - a resource manager for multi agent systems in smart spaces. In *CEEMAS '01: Revised Papers from the Second International Workshop of Central and Eastern Europe on Multi-Agent Systems*, pages 111–120, London, UK, 2002. Springer-Verlag.

[9] K. Gajos, L. Weisman, and H. Shrobe. Design principles for resource management systems for intelligent spaces. In *Proceedings of The Second International Workshop on Self-Adaptive Software*, Budapest, Hungary, 2001.

[10] P. Grace, G. Blair, and S. Samuel. A reflective framework for discovery and interaction in heterogeneous mobile environments. *SIGMOBILE Mob. Comput. Commun. Rev.*, 9(1):2–14, 2005.

[11] S. D. Gribble, M. Welsh, R. von Behren, E. A. Brewer, D. Culler, N. Borisov, S. Czerwinski, R. Gummadi, J. Hill, A. Joseph, R. H. Katz, Z. M. Mao, S. Ross, B. Zhao, and R. C. Holte. The ninja architecture for robust internet-scale systems and services373423. *Comput. Networks*, 35(4):473–497, 2001.

[12] R. Grimm, J. Davis, E. Lemar, A. Macbeth, S. Swanson, T. Anderson, B. Bershad, G. Borriello, S. Gribble, and D. Wetherall. System support for pervasive applications. *ACM Trans. Comput. Syst.*, 22(4):421–486, 2004.

[13] GSRC. Gigascale systems research center. http://www.gigascale.org/, 2006.

[14] J. Hill, R. Szewczyk, A. Woo, S. Hollar, D. Culler, and K. Pister. System architecture directions for networked sensors. In *ASPLOS-IX: Proceedings of the ninth international conference on Architectural support for programming languages and operating systems*, pages 93–104, New York, NY, USA, 2000. ACM Press.

[15] E. Kiciman and A. Fox. Using dynamic mediation to integrate cots entities in a ubiquitous computing environment. In *HUC '00: Proceedings of the 2nd international symposium on Handheld and Ubiquitous Computing*, pages 211–226, London, UK, 2000. Springer-Verlag.

[16] M. Lohse and M. Repplinger and P. Slusallek. An Open Middleware Architecture for Network-Integrated Multimedia. In *Protocols and Systems for Interactive Distributed Multimedia Systems, Joint International Workshops on Interactive Distributed Multimedia Systems and Protocols for Multimedia Systems, IDMS/PROMS 2002, Proceedings*, volume 2515 of *Lecture Notes in Computer Science*, pages 327–338. Springer, 2002.

[17] maemo.org. Maemo development plaform: White paper. http://www.maemo.org/platform/docs/maemo_exec_whitepaper.html, 2006.

[18] Z. M. Mao and R. Katz. Achieving service portability using self-adaptive data paths. *IEEE Communications Magazine*, 40(1):108–114, 2002.

[19] J. Nakazawa, H. Tokuda, W. K. Edwards, and U. Ramachandran. A bridging framework for universal interoperability in pervasive systems. In *ICDCS '06: Proceedings of the 26th IEEE International Conference on Distributed Computing Systems*, page 3, Washington, DC, USA, 2006. IEEE Computer Society.

[20] Nokia. Nokia 770 internet tablet. http://www.nokiausa.com/770, 2006.

[21] B. Phillips. Metaglue: A programming language for multi agent systems. Master's thesis, 1999.

[22] M. Roman, C. Hess, R. Cerqueira, A. Ranganathan, R. H. Campbell, and K. Nahrstedt. A middleware infrastructure for active spaces. *IEEE Pervasive Computing*, 1(4):74–83, 2002.

[23] M. Roman, F. Kon, and R. H. Campbell. Reflective middleware: From your desk to your hand. Technical Report UIUCDCS-R-2000-2195, University of Illinois at Urbana-Champaign, 2000.

[24] tinyos.net. TinyOS open-source website. http://www.tinyos.net, 2006.

[25] UPnP Forum. Understanding UPnP: A white paper. http://www.upnp.org/download/UPNP_UnderstandingUPNP.doc, 2006.

[26] VLC. Videolan client documentation. http://www.videolan.org/, 2006.

[27] M. Weiser. The computer for the 21st century. *Scientific American*, 1991.

Appendix A

DCP Grammar

The device control protocol (DCP) implements the hardware abstraction layer of our system architecture. This protocol is implemented using the LALR grammar below:

```
<message>:
( <messagebody> )

<messagebody>:
  register <registerbody>
  | render <renderbody>
  | ping <emptybody>
  | notify <notifybody>
  | reset <emptybody>
  | set_sinks <setsinksbody>
  | set_content_list <setcontentlistbody>
  | set_duration <setdurationbody>
  | get_sinks <getsinksbody>
  | get_sessions <getsessionsbody>
  | get_contents <getcontentsbody>
  | set_session_cap <setsessioncapbody>
  | set_cap <setcapbody>
  | get_cap <getcapbody>
  | get_session_cap <getsessioncapbody>
  | kill_sessions <killsessionsbody>
  | get_session_attr <getsessionattrbody>
```

```
<getcapbody>:
  req <capability>
  | ack <capability> <capinnerbody>
  | error <errorbody>

<getsessioncapbody>:
  req [0-9] <capability>
  | ack [0-9] <capability> <capinnerbody>
  | error <errorbody>

<setsessioncapbody>:
  req [0-9] <capability> <capinnerbody>
  | ack
  | error <errorbody>

<setcapbody>:
  req <capability> <capinnerbody>
  | ack
  | error <errorbody>

<caps>:
  <caps> <cap>
  | <cap>

<cap>:
  <capability>

<capability>:
  content_list
  | filename
  | destination
  | play
  | pause
  | stop
```

```
  | next
  | previous
  | volume
  | brightness
  | configuration
  | turnoff
  | intype

<capinnerbody>:
  [0-9]
  | [a-zA-Z]
  | ( <stringsandnumbers> )
  | ( <stringsornumbers> )

<errorbody>:
  [0-9] [a-zA-Z]

<registerbody>:
  req ( <signatures> ) ( <devicetypes> ) ( <caps> ) [a-zA-Z] [a-zA-Z]
  | ack [0-9]
  | error <errorbody>

<signatures>:
  <signatures> <signature>
  | signature

<signature>:
  ( <io> <mediatype> [0-9] )

<io>:
  in
  | out

<mediatype>:
  [a-zA-Z]
```

```
<devicetypes>:
  <devicetypes> <devicetype>
  | <devicetype>

<devicetype>:
  src
  | sink
  | rc
  | xcoder
  | sensor
  | compute

<renderbody>:
  req ( <stringsornumbers> ) ( <stringsornumbers> ) [0-9] [0-9]
  | ack [0-9] [a-zA-Z] ( <caps> )
  | error <errorbody>

<notifybody>:
  req [0-9] [a-zA-Z]
  | ack
  | error <errorbody>

<emptybody>:
  req
  | ack
  | error <errorbody>

<setsinksbody>:
  req ( <stringsornumbers> )
  | ack ( <stringsornumbers> )
  | error <errorbody>

<setcontentlistbody>:
  req [0-9] ( <stringsornumbers> )
```

```
  | ack
  | error <errorbody>

<setdurationbody>:
  req [0-9]
  | ack
  | error <errorbody>

<getsinksbody>:
  req
  | ack ( <strings> )
  | error <errorbody>

<getsessionattrbody>:
  req [0-9]
  | ack ( <stringsornumbers> ) ( <stringsornumbers> ) ( <caps> ) [0-9] [0-9]
  | error <errorbody>

<getsessionsbody>:
  req
  | ack ( <numbers> )
  | error <errorbody>

<getcontentsbody>:
  req
  | ack ( <strings> )
  | error <errorbody>

<killsessionsbody>:
  req ( <numbers> )
  | ack
  | error <errorbody>

<stringsandnumbers>:
  <stringsandnumbers> <stringandnumber>
```

```
  | <stringandnumber>

<stringandnumber>:
  ( [0-9] [a-zA-Z] )

<stringsornumbers>:
  <stringsornumbers> <stringornumber>
  | <stringornumber>

<stringornumber>:
  [a-zA-Z]
  | [0-9]

<strings>:
  <strings> <string>
  | <string>

<string>:
  [a-zA-Z]

<numbers>:
  <numbers> <number>
  | <number>

<number>:
  [0-9]
```

Appendix B

Setup and Running the Home Gateway Demo

B.1 Introduction

The logical structure of the home gateway demo is sketched in Figure B.1. There are three video clients, one audio client, one remote control, two content sources (audio and video), and a set of location sensors (floor mats with sensor motes). There is a central compute hub which performs the heavy lifting (transcoding and mapping), as well as, hosting a video client. The devices which participate in the demo are listed below:

- **XUP MPEG-4 Video** A client which can sink MPEG-4 compressed video. A software module runs on the FPGA to perform discovery and connection management. A hardware decoder implemented on the FPGA decodes the video to show on an attached monitor.
- **XUP Motion-JPEG Video** The same as the XUP MPEG-4 Video client, however, it takes Motion-JPEG formatted video.
- **XUP Audio** An audio client implemented on the XUP board; similar to other clients there is a software process managing connections and discovery; the on-board audio chip is used.
- **BEE2** A 5 FPGA board which performs the heavy-lifting. Transcoding is computationally intensive; an FPGA is allocated on the fly to support transcoding. On the

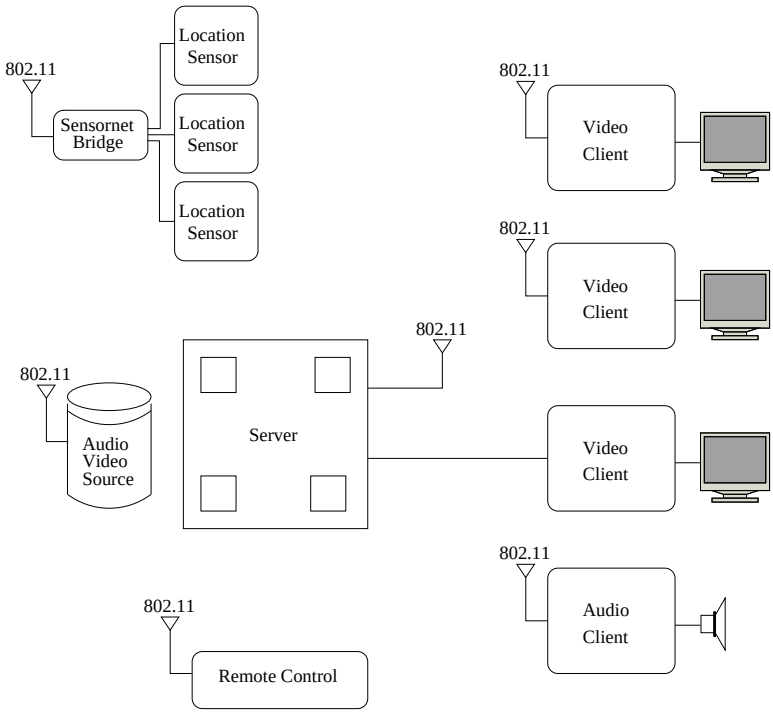


Figure B.1: Home Gateway top level setup.

central FPGA, the server runs on the PPC, managing the state of the network and performing routing of video streams.

- **Audio Source Mini** A Mac Mini running a software program which may stream data (push semantics) to an audio client.
- **Video Source Mini** Same as Audio Source Mini, however, it sources video.
- **Remote Control** Either a light-weight command line or GUI running on the Nokia wireless Internet PC. This device connects to the server and initiates the streaming between sources and clients.
- **Wireless-to-Ethernet Bridges** These bridges connect the XUP wired interfaces to the wireless network.
- **Router and Wireless Access Point** Responsible for the physical connectivity (wireless and wired) and IP routing of the network.

Table B.1: IP address table for clients and server.

Device	Client/Server	IP Address
XUP MPEG-4 Video	client	10.0.0.100
XUP Motion-JPEG Video	client	10.0.0.101
XUP Audio	client	10.0.0.102
BEE2	both	10.0.0.115
Audio Source Mini	client	10.0.0.110
Video Source Mini	client	10.0.0.111
Remote Control	client	DHCP

B.2 Networking

A router and wireless bridges are used to connect the devices of the demo using IP over Ethernet and wireless (802.11g).

B.2.1 Router Setup

We use a pre-n router from Belkin. The router's IP address is 10.0.0.1. The login is *admin* with password *sblig162*. Inside the router setup, you can monitor the wireless, DHCP, etc. The wireless SSID is *GSRC_DEMO* and it is secured with WEP key 2a20307b2f37d2e077c87f11cf. Addresses are in the subnet (Class A) 255.0.0.0; DHCP addresses are assigned within: 10.0.0.{128–254}.

B.2.2 Wireless Bridges

Wireless bridges are configured to connect to the router and obtain a DHCP address automatically. The router's DHCP client list gives the bridge IP addresses. The bridge administrative web pages can be accessed with an Internet browser pointing at their IP addresses; the login/password is *admin*/*admin*.

B.2.3 Client/Server IP Addresses

Clients and the server do not obtain their IP addresses dynamically (except for the remote control); instead they are hard-coded so that the software they run can make this simple assumption. Table B.1 gives the IP addresses for the network clients and server:

B.2.4 Connections

All XUP clients and the BEE2 are connected to wireless-to-wired bridges. Audio and video sources are connected directly to the router through Ethernet. The location sensors (floor mats) are each connected to a Mica2 sensor mote. A sensornet-to-wireless bridge is implemented through a one of the Mac Mini devices. Finally, the remote control is connected through it's wireless interface (Nokia device).

B.3 Source Code

This section will show where and how to get the source for building the demo; it will not discuss the architecture or design.

B.3.1 Location

The software cvs repository is located on hitz; the directory is called *HomeGateway*. To modify the repository (add and checkin), you need to be part of the *gateway* group. In Linux, set the following variables and checkout the *home_gateway* module from cvs. For information on using CVS, see Google.

```
export CVSROOT=sunv40z-1.eecs.berkeley.edu:/tools/designs/HomeGateway
export CVS_RSH=ssh
```

The source code is located in `home_gateway/src`; the directory tree is shown below. The important directories are `clients` (containing the remote control, audio/video clients, content sources, and common library, `clientlib`), `dcp` (device control protocol to stitch all the devices together), `server` (management of devices, state, routing, etc.), and `util` (contains some wrappers around common functions).

```
src/
|-- clients
|   |-- clientlib
|   |-- file_sink
|   |   |-- mmm
|   |   `-- slideshow
|   |-- file_source
|   |   |-- mpg4_support
|   |   `-- test_file_source
```

```
|   |-- remote_control
|   |   |-- cmd_line
|   |   `-- gui
|   |       |-- data
|   |       |   `-- icons
|   |       |       |-- 26x26
|   |       |       |-- 40x40
|   |       |       `-- scalable
|   |       |-- debian
|   |       |-- po
|   |       `-- src
|   |           `-- ui
|   |               `-- appview
|   |-- sensor
|   |   `-- hg_mat
|   |-- xcoder
|   `-- xcoder2
|-- dcp
|   `-- parser
|-- server
|-- server_stub
|-- test
`-- util
    |-- monitor
    |-- util
    `-- wrappers
        |-- lib
        `-- libfree
```

B.3.2 Compiling Software

Currently, the compilation of the software is not automated. The tool prerequisites are:

1. gcc
2. make

3. flex/bison
4. libjpeg62-dev
5. libreadline (should come with libhistory)

Under Linux, user your favorite package manager (*e.g.* yum or apt-get) to install these programs. Each of the following subsections will detail how to build the DCP library, logging infrastructure, clients, and server. Each build should be done in this order.

B.3.2.1 Wrapper Functions

We have wrapped a set of common functions such as `malloc()` and `free()` to help with debugging and maintenance. First, descend into the `src/util/util` directory and type `make` to build the *libhgutil.a* library.

B.3.2.2 DCP

Next, we will compile the device control protocol (DCP). Compiling the DCP library (and parser) is accomplished by descending to `src/dcp/` and typing `make`. This will build a library file called *libdcp.a*.

B.3.2.3 Logging

To compile the logging infrastructure, go to `src/util/monitor` and type `make`. This will build the application *log_listener*. This listener application will capture log messages from clients and the server during their execution and output them to a file. This data can be helpful for debugging.

B.3.2.4 Clients

To build the clients, first we need to build the client library. The source code for this is found in `src/clients/clientlib`. Typing `make` in this directory will build the library (used by all clients for DCP functionality) called *libclient.a*.

Next, for the client directories `file_sink`, `file_source`, and `remote_control`, descend in to each directory and type `make`. Each will build an executable who's operation will be described further in the next section.

The GUI version of the remote control runs on the Nokia 770 device. To build the application for this device requires a much lengthier build process and is beyond the scope of

this document. For more information on building this GUI application refer to the Maemo website (`http://www.maemo.org`).

B.3.2.5 Server

Finally, descending into `src/server` and typing `make` will build the server executable. This build process may take a minute or two so be patient. If you have a dual core processor we recommend using the `-j3` option.

B.4 Running Software

Once the software is compiled, there are several simple steps to get the clients running and connected to the server.

B.4.1 Logger

The logger is a program to listen for connections from clients and log their debug messages to a file. It can be started using the following command. Replace the listen port with the port used by each of the clients.

```
./log_listener -p <LISTEN PORT>
```

B.4.2 Clients

Below are descriptions for starting the demo clients. All clients can take an additional optional argument specifying the IP address and port of the logger in the form of `net:<IP ADDRESS>:<PORT>`.

B.4.2.1 XUP Video

The XUP video client can be started with the following command line. Server IP address and port must be specified depending on the location of the running server. Currently the server listens to port 50000. The name and location are optional but recommended. The name is a string without spaces, while the location is an integer indicating the virtual room number.

```
./file_sink net:<SERVER IP ADDRESS>:<SERVER PORT> -n <NAME> -r <LOCATION>
```

for example:

```
./file_sink net:127.0.0.1:50000 -n file_sink2 -r 2
```

B.4.3 Remote Control

The command line version of the remote control can be started in a similar manner to the video clients. A server IP address and port must be specified.

```
./remote_control net:<SERVER IP ADDRESS>:<SERVER PORT>
```

B.4.4 Content Sources

Similar to other clients, the content source software running on the Mac Minis can be started using the following command:

```
./file_source net:<SERVER IP ADDRESS>:<SERVER PORT>
```

B.4.5 Location Sensors

Turn on the Mac-mini and make sure that the mica2 mote with TOSBASE is plugged in to the mini. The mote must be plugged into a serial-usb converter and may be plugged into a usb hub (as long as it is a usb 1.x hub, and not 2). In the `/home/gsrc/tinyos-1.x/` directory source setupenv.sh. Make sure the TOSBASE mote is turned on and that it has registered as `dev/ttyUSB0` (this is the default if only one mote is directly connected with a serial-usb converter, but may change if you connect many nodes to the mini.) Change the directory to `/tinyos-1.x/tools/src/sf` and execute (where 50000 and 10.0.0.110 are the port & address of the server):

```
./sf 50000 10.0.0.110 /dev/ttyUSB0 57600 mica2
```

B.4.6 Server

The server software running on the BEE2 [4] can be started by simply typing the following command:

```
./server
```

Other clients will automatically connect to the server. Note that the IP address which is assigned statically, must be used in the command line arguments for all other clients.

Currently there is an option to also use the BEE2 as a transcoding client. This can be accomplished by using the same XUP Video command line specified above. However,

before doing so, the proper drivers must be initiated on the server. This can be done by executing the `./startdvi` script.

B.5 Controlling Demo

Once all the devices are networked, the code is downloaded and compiled, and the client and server programs are running, one can use the following set of commands to control the demo. This description will assume the use of the command line remote control. Consider the GUI remote control an easier to use version of the command line version – all the same concepts still apply but they will be accomplished with buttons, lists boxes, etc.

B.5.1 A Simple Session

All sessions begin with the `render` command. A simple example is the following:

```
(render r ("video1.bit" "video2.bit") ("file_sink1") 1 0)
```

The playlist contains two files, "video1.bit" and "video2.bit", and one sink device, "file_sink1". The "1" denotes the start position in the playlist (1-based) and the "0" gives the duration in seconds to play each of the files in the playlist. A value of zero indicates that the files of the playlist have their own duration (as video and audio files do) and their implied duration will be used to increment the index when they are finished playing. Photos do not have an implied duration, so a value of "5" would show each photo in the playlist for five seconds.

B.5.1.1 Streaming Audio/Video

Once the render command has been executed, the user can set the capability of the session to `play`. This is done by typing:

```
(set_session_cap r 0 play 0)
```

The first "0" is the session number (which will be returned to the remote control after the `render` command is executed successfully. The "`play`" capability tells the session to begin streaming the first video. The final value (a number) is ignored.

If the user wants to play the next file in the playlist, they typing the following will accomplish this task:

```
(set_session_cap r 0 next 0)
```

Similarly, if they want to play the previous file in the playlist, then executing the following command will do the trick:

```
(set_session_cap r 0 previous 0)
```

Finally, stopping the streaming of data can be performed using the following command:

```
(set_session_cap r 0 stop 0)
```

B.5.1.2 Killing a Session

When one wants to destroy a session, they can use the `kill_sessions` command in the following manner:

```
(kill_sessions r (0))
```

The "(0)" is actually a list of session identification numbers. If there were multiple sessions, then this may be replaced by "(0 2 6 15)" to kill sessions 0, 2, 6, and 15.

B.5.2 Advanced Sessions

There are more advanced features of the session such as using multiple sources and sinks, and location awareness.

Using multiple sources is implied by specifying a playlist which contains file names that exist on different devices. In the example above, the different .bit files may exist on different devices. In this case, system will take care of re-mapping the proper device.

Using multiple sinks is specified in the `render` command in a similar manner to the playlist. Multiple sink names are listed, such as:

```
(render r ("video1.bit" "video2.bit") ("file_sink1" "file_sink2") 1 0)
```

In this case, the video stream "video1.bit" will be streamed to two sinks simultaneously. One additional step beyond this is to instead of specifying each sink name explicitly, specify a generic name for a sink and stream to a device which is nearest the user. This can be accomplished with the following render command:

```
(render r ("video1.bit" "video2.bit") ("video") 1 0)
```

Here, the "video" keyword specifies a video renderer which will be mapped at run-time based on the current location of the user. All the same session controlling commands above still apply to these more advanced sessions.