Task-Driven Computing

Zhenyu Wang David Garlan

May 2000
CMU-CS-00-154

School of Computer Science
Carnegie Mellon University
Pittsburgh, PA 15217

Abstract

We are moving towards a world of pervasive computing in which users can access and manipulate information from anywhere at anytime. Computing devices and networks are becoming ubiquitous. In this new world, computing will no longer be tethered to desktops: users will become increasingly mobile. As users move across environments, they will have access to a dynamic range of computing devices and software services. They will want to use the resources to perform computing tasks. Today's computing infrastructure does not support this model of computing very well because computers interact with users in terms of low level abstractions: applications and individual devices. Today, if a mobile user wants to use the computing resources of a new environment, he has to manually figure out how to perform a computing task using local resources and/or to migrate his computing context from another environment. Such manual operation is unacceptable in a pervasive computing world because it does not scale with the increasing amount of software services, user mobility and resource dynamism. It also demands that users spend too much time on non-task related configuration activities. This technical report describes an approach called task-driven computing that can be used to solve this problem. The approach is based on the insight that with appropriate system support it is possible to let users interact with their computing environments in terms of high level tasks and free them from low level configuration activities.

This research was sponsored in part by the Defense Advanced Research Projects Agency under agreement numbers F30602-97-2-0031 and N66001-99-2-8918. The U.S. Government is authorized to reproduce and distribute reprints for Governmental purposes notwithstanding any copyright annotation thereon. The views and conclusions contained herein are those of the authors and should not be interpreted as necessarily representing the official policies or endorsements, either expressed or implied, of the Defense Advanced Research Projects Agency Rome Laboratory or the U.S. Government.

Keywords: tasks, mobile computing, pervasive computing, service discovery, human-computer interaction

1 Introduction:

We are moving towards a world of pervasive computing in which users can access and manipulate information from anywhere at anytime. Computing devices are becoming ubiquitous. Not only are desktop computers everywhere, non-PC devices (such as set-top boxes, screen phones, networked displays, networked cameras, networked speakers, remote input devices and intelligent watches) are moving into the main-stream marketplace. Networking is also becoming pervasive with the increasing popularity of home networking and wireless overlay networks (HomeRF, Bluetooth, CDMA, wireless ethernet etc.).

In this world, computing will no longer be tethered to desktops: users will become increasingly mobile. As they move across environments, they will have access to a dynamic range of computing devices and software services. They will want to use the resources to perform computing tasks, such as writing documents, reading email, getting reminders and accessing information.

For example, suppose that I attend a conference or visit a lab and that I have a laptop with me. I discover that there is a UNIX workstation on-site that I can use. Instead of having to find a phone jack and use the modem to check my email with the laptop, I would like to use the workstation because it has a faster connection and the necessary document viewers for displaying email attachments. If the workstation is unable to handle a particular attachment in MS Word format, then I want the system to display the attachment on my laptop, which has a Word viewer on it.

Moreover, as people move from one environment to another, they should be able to quickly migrate their computing context. Computing context includes the state of computing tasks, such as currently open documents and editing positions, open email messages, application options such as POP settings, personal data. The illusion of “continuity” across environments is important when you have long-lasting tasks and highly mobile users. When a mobile user comes into a new environment, he wants to simply resume his tasks and use currently available resources to keep working.

For example, a mobile real estate agent may start some applications in his office using local servers and graphics workstations to show a list of houses to a customer, then move to the car with an automobile-based device, and then a hand-held device when inside a house.

Today’s computing infrastructure does not support this model of computing very well because computers interact with users in terms of low level abstractions: applications and individual devices. Today, if a mobile user wants to use the computing resources of a new environment, he has to manually figure out how to perform a computing task using local resources and/or to migrate his computing context from another environment. Specifically, in a typical scenario he needs to

1. keep track of his computing context (usually mentally),
2. find out what computing resources he has access to in the new environment,
3. find out what kinds of software services/applications are installed and which should be used to realize a computing task,
4. configure the chosen services to reestablish his computing context, often involving setting application options, moving and transforming necessary files, initializing software services and coordinating multiple computing devices

In the laptop example above, first I would need to locate the local workstation that has a high speed network connection. Then I would need to figure out if there is a POP client installed on the workstation. If not, I must telnet back to my office machine and check my email remotely. If I get a message with a postscript attachment, I would then need to check if a X server is available on the workstation and try to set up my X environment. If the workstation refuses external X connections for security reasons, then I will not be able to display the attachment remotely. In that case, I will need to see if there are any document viewers installed locally. Assuming I find a local PDF reader, I have to move the attachment over using ftp, find out if there is any document conversion service that translates postscript into PDF, convert the file, then use the local viewer to display the attachment.

Such manual operation is unacceptable in a pervasive computing world for three reasons.

1. *It does not scale with increasing numbers of services and computing devices*. The knowledge and time required of users today will increase dramatically as more software services and computing resources become available.
2. *It does not scale with increasing user mobility*. Manual configuration costs time. If a user remains in a computing environment for only 15 minutes, he does not want to spend the first 10 minutes restoring computing contexts manually.
3. *It does not tolerate change or failure of computing resources*. The services available in an environment change in the presence of mobile devices, such as laptops and I/O devices. Software services can be installed or removed without users' knowledge. Approximity-based networking (such as IrDA and Bluetooth) leads to dynamically changing networks. Failures of services and networks can change the availability of computing resources. Manual configuration is simply too costly in this setting of transient services, networks and devices.

With the exponential increase in the power of computing devices, the most precious resource in a computer system is shifting from hardware resources to user attention. As noted, in today's world much of a mobile user's attention is wasted on low-level activities such as managing groups of computing devices, mentally keeping track of computing context and manually reinstantiating it, and figuring out how to accomplish a task using available services.

In this paper, I describe a new approach called Task-driven Computing to address these problems. The key idea behind the approach is that the system should take over many low-level management activities so that users can interact with a computing system directly in terms of high-level tasks that they wish to accomplish.

As I will detail later, this new approach centers on following elements:
1. System-maintained, globally-accessible abstract computing tasks and task states.
2. New types of services (or wrappers on legacy services) that support automatic configuration.
3. Automatic selection and configuration of service coalitions to accomplish computing tasks and reestablish computing contexts.
4. Proactive guidance to help a user accomplish a task and compensate for resource limitations in an environment.
5. Automatic management of dynamic service and resource conditions.

Making computing contexts globally accessible frees users from having to remember them mentally. Automatic collection of environment information frees users from having to locate accessible services in an environment by hand. Automatic task-based service discovery and configuration makes rapid configuration possible, even in the presence of large dynamic collections of devices and resources. The new approach will significantly reduce the attention and knowledge requirements of mobile users, and make possible automatic adaptation to change and failure. If fully realized, a user would be able to walk into any environment and have all services configured automatically so he can resume his computing tasks. He need no longer care about issues such as data location, data types, applications and configurations.

This paper describes what task-driven computing is, what technical challenges it poses and some possible solutions to these challenges. It also describes the current status of research on task-driven computing in the Aura project at Carnegie Mellon University. In Section 2, I give a background overview and explain in detail the concept of task-driven computing. After that, I compare task-driven computing with existing work. Then in Section four, I elaborate on the specific technical challenges of task-driven computing. After that, I give an overview of my approach to address those challenges. In Section six, I discuss some implementation issues.

2 Overview:

2.1 Motivation:

Today's computing model is characterized by the following assumptions:
- *Desktop computing*. People typically sit in front of a desktop PC to do their work.
- *Private device, private software*. Typically people own some stationary devices such as desktop workstations and a couple of mobile devices (laptops, PDAs), and they do their computing primarily on those devices. People have

complete control over those devices and how they are configured. People buy software and install it on their personal devices and compute primarily with that software.

- *Monolithic applications with human interface only*. Software applications are monolithic units (implementing many features) primarily designed to interact with humans instead of with other applications.
- *Manual mapping of computing tasks to applications*. Users must keep track of how to use these applications and configure them to carry out each task that they want to do.
- *Single device computing*. Users typically only use one device at a time.
- *Manual configuration*. Users are responsible for configuring applications.

In recent years, we have seen an explosion of networkable computing devices such as PCs, laptops, PDAs, information appliances, thin terminals, set-top boxes, net cameras, IP speakers, IO devices, screenphones etc. Networking is becoming ubiquitous with the wide acceptance of long-range and short-range wireless networking, home networks and increasingly available broadband connections such as DSL and cable. With these driving factors, all of the above assumptions are challenged.

- *Computing is becoming highly mobile with pervasive computing and networking*. Users have access to computing resources in a variety of environments (car, office, home, airport etc). Computing is not bound to the desktop anymore.
- *People have access to more computing devices than ever before*. Users also have access to more public devices such as servers and public displays.
- *Monolithic applications are being replaced by collections of cooperating services*. Those services are designed to interact with each other, not just with end users.
- *Mappings of tasks to services are dynamic*. There are different computing resources in different locations. These computing resources can be used to accomplish users' computing tasks to varying degrees.
- *Multiple computing devices can work in harmony*. With new industry standards such as Jini, Salutation and UPnP, computing devices can connect to each other automatically.

With all these new characteristics, one thing remains unchanged. That is users are still supposed to assume full responsibility for managing their computing environments to accomplish their computing tasks. Until recently, this has not been a problem since users have been largely stationary and have only had to deal with a few big applications on a small number of devices. However as pointed out in the introduction, with the increase of computing devices, software services and user mobility, this responsibility distracts users from accomplishing computing tasks effectively and does not scale as the number of devices escalates.

Ideally, as users change environments, they should be able to quickly take advantage of local resources to continue working on computing tasks without a lengthy manual set-up process. This entails three objectives,

1. the system should be able to resume *old* tasks, taking advantage of available resources,
2. the system should be able to automatically use local resources to accomplish *new* tasks,
3. the system should minimize disruption to a user in the presence of changing resources.

2.2 Approach – Task-driven Computing

Task-driven computing represents a new view towards system building. The basic observation here is that with explicit representation of computing tasks and disciplined ways to build software services, it is possible to build middleware infrastructures that enable automatic instantiation of task context into a collection of dynamic computing devices with a minimum amount of user interference. Software services should be built with the inherent expectation that they are going to be used with other services to achieve high-level goals. Users should interact with computing devices in terms of high-level tasks instead of individual services or applications.

Task-driven computing is based on two basic concepts: tasks and sessions (and interactions between these two elements). A task represents a user's computing intention. A task may range from simple activities such as document editing, to complex activities involving multiple steps, such as arranging a trip to Paris. Some tasks are short-lived, such as playing a movie. Some tasks are long-lived, such as selling a house. A task is platform-independent and application-independent. A session represents a group of collaborating computing devices (and services running on them) that are used to accomplish a task. In the example described below, a document browsing service, a VRML viewer, a financial application, an email client, and a notification service work together in a session to realize part of a house-selling task. A device can join or leave a session at will.

In more detail, task-driven computing consists of the following elements, most of which are illustrated in the example that follows:

E1: Application/environment-independent specification of tasks. A language is provided for task specification.
E2: Explicit management of computing context, which includes task specifications, task states and user preferences. A user's computing context is globally accessible and updated automatically as progress is made on tasks.
E3: Services with well-defined interfaces and meta-data specifications. Services are required to have certain interfaces that support automatic configuration. Legacy applications can be wrapped to support these interfaces.
E4: Session management: automatic service configuration, composition & coordination. Services are automatically composed and configured to accomplish user tasks. The system automatically adapts service configurations when service availability changes.
E5: Proactive guidance. When a task can't be executed because of resource restrictions, the system offers suggestions to the user to help him accomplish what he wants to do.

To illustrate how the system suspends a task in one environment and reestablishes it in another environment without user intervention, let us consider how task-driven computing works in a mobile real estate agent scenario. Suppose the task is "showing a house to a customer and letting me know if another customer has decided to buy the house." An agent might create the task in his office. To start the task, the system notices the existence of a VRML viewer on a graphics workstation, various document editors, financial applications, email clients, and windows pop up service. The system displays the brochures in the document editor and uses the VRML viewer to show the house currently selected by the customer. Customers can also use the spreadsheet service to calculate mortgage payments. When mail arrives showing that another customer has decided to buy the house, the system will pop up a window on the workstation to notify the agent. Now suppose the agent moves into his car. The task follows. Before the user leaves his office, the system collects the states of the task such as list of houses, current selections, different payment plans. As he enters the car, the system notices the existence of an auto PC with internet connection, a car phone, and a handheld Windows CE. The system configures the automobile PC for email and installs mail triggers, when a message arrives, the system will have the auto PC automatically dial the car phone for notification. Noticing that the CE device is only capable of handling 256 color static images and text documents, the system automatically pulls data from the office computer, makes necessary conversions and sends them to the CE device so the agent or the customer can continue browsing house information in the car.

2.3 High-level Challenges

From a technical perspective, the basic challenges associated with task-driven computing can be classified into the following categories:

- **Task Management**: We need mechanisms to describe users' computing intentions. Also we need explicit representation mechanisms for task states (E1 & E2 above)
- **Service Construction & Encapsulation of Legacy Systems**: Services must be packaged in certain ways and support certain interfaces to fully participate in the task-driven computing world (E3)
- **Session/Service Management**: We need mechanisms to choose, configure and coordinate software services (E2, E3, E4, E5).
- **Environment Management**. We need mechanisms to automatically collect information about available services and devices and to monitor resource changes.

This paper primarily describes service management (E2, E3, E4, E5) and some of the task management issues (E1). Existing work on service discovery (such as Jini or uPnP) and network/OS resource monitors is addressing issues related to environment management. Most of E1 is being addressed by other research communities such as robotics planning, agents, etc.

3 Comparison with Related Work

Existing work on supporting mobile users as they move across physical environments falls into the following three general categories: mobile computing, service discovery and composition, and process-oriented computing.

3.1 Mobile Computing

One approach to support mobile computing [13,16] is using a thin-client computing model, represented by web-based applications and thin network devices such as SunRay [17] or X Terminals. Since all user-computing context is stored on a server and users use only standard client devices to access the services, users can easily suspend their computing tasks in one environment and resume them in another.

This approach is not suitable for task-driven computing because of two limitations. First, with this model, users can't take advantage of local services since all computation is carried out in backend servers. A high bandwidth network connection is required between the servers and the client devices. Second, this model assumes that people will always use similar clients (Sun Ray enterprise appliance or X server) to access backend services. Task-driven computing addresses both of these limitations.

Another approach to support mobile computing is using portable computing devices. In this model, a user uses a portable device for most of his computing needs. With advancements in both hardware (reduction in power consumption and size, better screen) and software (e.g. distributed file systems [11]), users can take a mobile device and perform his tasks almost anywhere. This approach does not take advantage of computing resources available in a user's surrounding environment.

There is also some existing work on automatic application for nomadic users. Remote configuration mechanisms such as the Application Configuration Protocol (ACAP)[15] allow individual applications to access remotely-stored options and user preferences such as bookmarks and address books and configure themselves automatically. But this is far from sufficient. First, ACAP is not environment sensitive. As a simple example, ACAP can't automatically set up a X connection. Second, ACAP is for applications only, it does not configure a collection of services to fulfil a task such as I/O hookup, transcoders hookup, data type transformations. For example, suppose a user is reading email on a UNIX machine that can't handle word attachments. If there is a laptop nearby, the system should ideally automatically copy the attachment to the laptop and display it there. This is beyond the capability of ACAP.

3.2 Service Discovery and Composition

There has been a lot of work on dynamically locating software services, ranging from early efforts such as CORBA[9] and COM[10] to the more recent service discovery frameworks such as Jini™ [6] and Universal Plug & Play (UPnP™ [7]). These mechanisms allow a client to dynamically locate and access software services. However, these frameworks only work at an individual service level. Task-driven computing builds on these low-level efforts. It delivers high-level protocols and mechanisms to truly free users from having to think about non-task related issues such as keeping track of computing states, instantiating tasks and dynamic reconfiguration.

Many researchers have looked at the issue of software composition (sometimes called meta-programming[1,2,3]). Formal languages have been developed to better enable specification of component interfaces crucial to component composition. Software component frameworks (such as Javabeans[11] and COM[10]) allow developers to create software systems out of pre-built binary components. Domain-specific component integration frameworks (such as HLA, IBM SanFrancisco, Enterprise JavaBeans, etc.) have been developed to support software component composition in specific domains. Task-driven computing builds on this work. The main technical challenges task-driven computing addresses are not centered on *how* to compose software services. Instead, the challenges focus on task management, *what* services to compose and dynamic service management.

There has been some HCI work on integrating multiple computing devices. Project Pebbles[5] demonstrates that PDAs can collaborate with PCs using ad-hoc mechanisms. Task-driven computing takes such efforts a step forward by delivering a general framework that enables the collaboration among arbitrary computing devices.

3.3 Process-oriented computing

There exist many process languages such as workflow specification languages [18]. Using these languages, a user can create high level computing tasks in a script-like form. The focus of task-driven computing is not investigating the pros and cons of different process languages. Rather, task-driven computing can be used with many process languages. As noted later, in this work, a low-level task specification language can be designed that other process languages can be compiled into.

4 Detailed Technical Challenges:

In this section, I elaborate on specific technical challenges in each of the categories outlined in Section 2.3 (In the next section, I will explain how these challenges can be addressed.)

4.1 Task Management

Representation of tasks. In order for computers to process tasks, we have to be able to represent tasks. A task specification language must let you talk about flows and primitives. Task flows decompose a complex task into a sequence of steps (subtasks or primitives). A task primitive is a unit of action that has to be carried out by the actual software services.

C1: Management of task primitives. A task primitive is the basic unit of a task that can be instantiated and composed with other primitives. It should be application-independent to be instantiatable in heterogeneous environments. Application independence also frees users from having to think about low-level details. Questions that arise are: What is the appropriate representation for a task primitive? How should the system encode the semantics of a task primitive so it can be instantiated dynamically? When and how can a new type of new primitive be added to the system?

C2: Task primitive composition. What kind of task flow language should be used to compose the primitives?

C3: Lightweight task creation. Many task definition languages (TDLs) are very powerful. However, they are too heavy weight for ordinary users to use everyday because they require users to program using a special language. It is crucial to make it easy for users to create tasks. Lightweight task creation can be enabled in many ways such as watch-me macros and customizable templates. The most interesting approach is to use system snapshots to create tasks. The main issue here is that system snapshots are application-specific and platform-dependent. How can we extract enough application-independent information from snapshots so it can be instantiated in another environment using a different set of applications?

Representation & management of task states. As users move across environments, we want their computing state to "follow" them. In other words, when a user leaves a computing environment, the current task states need to be retrieved. When he enters a new environment, a user's task states must be quickly propagated.

C4: Formal representation of task state. Task states include the following major elements: task-related configuration information, task-related data and computing state. Other information might also be associated with task state such as task instantiation history. Task-driven computing must address the following question: How should the system organize the task state to enable efficient retrieval and propagation across environments? As argued below, for performance reasons, the representation of task states must allow for structured and incremental updating.

C5: Retrieval and propagation of task states. The key issue here is what kinds of interfaces should services provide to enable efficient retrieval of a consistent task state. When a task is re-instantiated in an environment that the user has visited before, it is possible that some of the physical services will still have the old task state in their cache. We want to be able to take advantage of this to improve performance.

4.2 Service Management

As mentioned in the overview section, tasks are realized by service coalitions. Service management serves three purposes,

1. to choose, configure and initialize services in order to resume old tasks or start a new task. For example, it should be possible to use the local timing, calendar, and audio services to realize a reminder task.
2. to coordinate the collaboration of multiple services/devices. For example, it should be able to use a PDA as a remote controller of desktop applications.
3. to reconfigure services to maintain task integrity in the presence of variable resources. For example, when a WinCE device comes into the infrared communication range of a PC, the system might shift the user input focus from keyboard to the stylus.

The major challenges with service management are to design appropriate interaction protocols between services and task management infrastructure that accomplish the above goals, and to design appropriate service interfaces that facilitate these protocols.

<u>Task Instantiation</u>

A Task Instantiation Protocol (TIP) is needed to instantiate a task as a group of services. If a direct task instantiation fails, the system should offer tips to help users correct the situation.

Part of TIP provides service discovery functionality similar to what is provided by OMG Trader, Jini or the Service Location Protocol. However, in addition, it must also provide these extra capabilities:

- Service configuration. Services need to be configured and properly initialized.
- Service composition. Services may need to be composed to accomplish a task. The system should create communication channels and provide adapters if necessary.
- Proactive guidance. In the case where a task can’t be accomplished because of (temporary) resource limitations, the system should offer hints to users on how to accomplish the task.

C6: Service location and composition. How should the system represent the semantics of services so that the system can choose the appropriate services? What algorithm can be used to locate all necessary services to instantiate a task? How do services communicate with each other? How should the system initialize the services so old tasks can be resumed?

C7: Proactive guidance. As explained earlier, the system should offer suggestions on how to accomplish a task when the task can’t be instantiated directly. The following two questions need to be answered: First, what kinds of guidance can be offered? Second, what kinds of mechanisms can be designed to support the guidance?

<u>Task State Management Protocol (TSMP)</u>

TSMP is used for efficient retrieval and propagation of task states (working with TIP).

- Retrieval: when a user leaves an environment, a consistent snapshot of the services must be retrieved so the task can be instantiated later in another environment. Also, before a collection of services is voluntarily detached from the network (and thus not accessible anymore), their states should be collected so the system can reconfigure and reinitialize the remaining services to resume the tasks.

- Propagation: when a user enters an environment, and when a collection of services becomes available, the task state must be propagated into the newly available services.

C8: Task state retrieval and propagation. What kinds of support are needed from services? How should the system obtain a consistent state snapshot? How does one support efficient, incremental retrieval and propagation?

Service COOrdination Protocol (SCOOP):

SCOOP is used to maintain task integrity in the presence of changing resources. The availability of services may change for a variety of reasons, such as device mobility, ad-hoc network connections, network outage and resource contention. When services become available or unavailable for a task, appropriate steps must be taken to ensure that the task can continue with a minimum amount of user intervention, and yet at the same time take advantage of the newly available services.

C9: Service coordination. The main issue here is how to dynamically modify the system configuration.

5 Plan of Approach

My approach to addressing the challenges enumerated above is based on two basic ideas. First, tasks are composed from application-independent primitives that are bound to actual services in the environment. Second, service dynamism and environment dynamism are managed by appropriate protocols.

5.1 Virtual Services

The key element of task management is the concept of virtual services. Denoted by a type, a Virtual Service (VS) represents a unit of abstract functionality, such as the ability to display a document or to accept user input. The functionality of a virtual service is defined by its interfaces. Virtual services can be composed using a flow language to specify a task.

In order for a virtual service to be invoked, it must use the actual resources available in an environment. We model both computing devices and the software running on them as *physical services*. A virtual service is realized by physical software services through a process called *binding*. Multiple virtual services can bind to a single physical service. For example, document editing and spell checking can both be bound to Microsoft Word. As discussed later, a virtual service can also be bound to multiple physical services. For example, a cell phone and a PC running Outlook Express can both bind to a phone book service.

To elaborate, the main characteristics of a virtual service are:

- **A virtual service type denotes a capability.**

  The interfaces of a virtual service include four elements: a functional interface, a state interface, a configuration interface and a dependency specification.

  The functional interface of a virtual service defines how a client can access the service. It can be described using an ordinary interface definition language (such as CORBA IDL) in the forms of communicating messages, method invocations, and data pipes.

  The dependency specification can be used to specify other services required by a virtual service in order to operate. For example, a POP client may require that a text editor be available.

The state interface describes the state constituents of a virtual service. State is defined using a structure schema (such as an XML schema) that specifies the state of a service in terms of attributes and values[1]. For example, the state schema of a POP client might include attributes such as current mailbox, current messageID, email address shortcuts, etc. The state schema of a document display service will include the name of the document, the current position in the document, and possibly some display options.

The configuration interface describes the configuration parameters of a virtual service. It is also represented by a structure schema. For example, a POP client configuration interface might specify options, such as the mail server name; a document editing service may include options such as the auto-save interval and tab size.

- **Virtual service types are organized in an inheritance hierarchy.**

  Virtual service types form a hierarchy. The functionality of a parent must also be delivered by its children (or subtypes). One service specializes another service if it delivers more functionality or provides finer control over its operation. For example, part of a virtual service hierarchy may look like this:

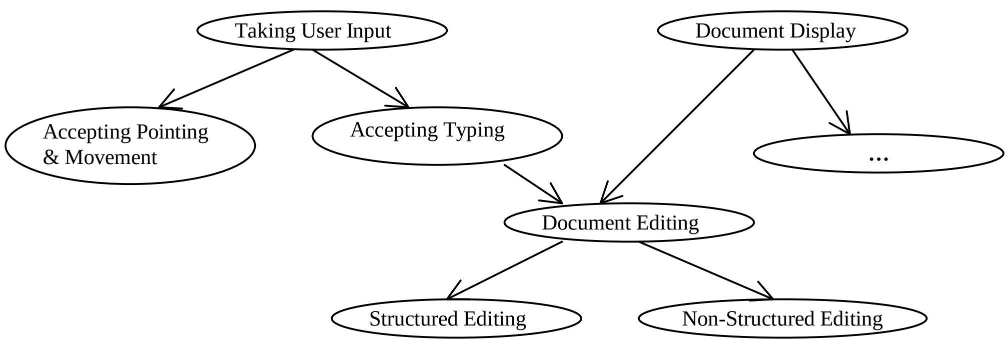


Figure 1: Virtual Service Hierarchy Example

The state and configuration interfaces of a parent service must also be inherited by any child service, which means that the attributes in a parent schema must also appear in the child schema. The dependency works in the opposite direction: That is, any item in a child’s dependency list must also appear in its parent’s dependency list.

There are two reasons for organizing the type system in this way: First, the number of virtual services can be quite large and there must be some way to organize this large space. Second, users can express a computing intention at different abstraction levels representing varying degrees of commitment to the details of how a virtual service should be controlled, what features it should support, etc.

- **A virtual service may be decomposed into a group of interacting services.**

  A virtual service may itself be composed of a group of collaborating services. For example: a reminder service may be composed of a timing service, a calendar service and a notification service. A properly designed scripting language can allow a user to compose services using a set of operators including sequencing, parallel execution, conditional branch and interruption.

- **A task is a top-level virtual service.**

  A task is simply a top-level virtual service that can be invoked directly by a user.

[1] There are three basic reasons for using structure schemas: ease of incremental updating, ease of inheritance, ease of name standardization.

5.2 Representation of Run-time Tasks

At run-time, a task is represented by a session. Logically, a session consists of a set of cooperating virtual services with each "bound" to one or more physical services in the environment (Figure 2). Physical services interact with each other using events, method invocation and pipes prescribed in the task script. A virtual service can voluntarily or non-voluntarily join or leave a session.

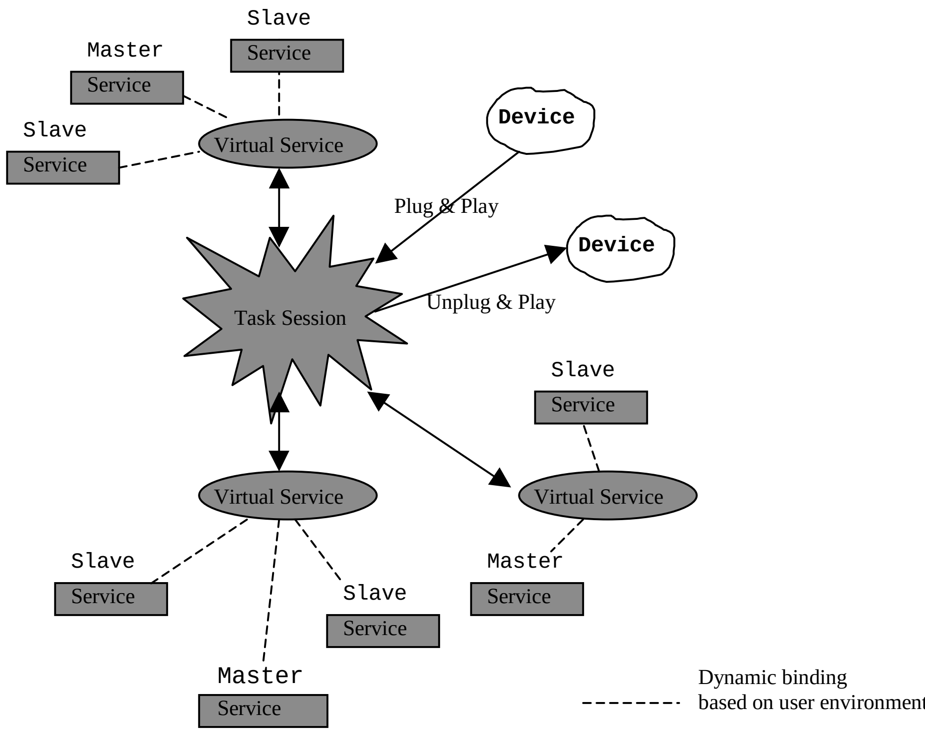


Figure 2: Session and Binding

To elaborate, the key points here are:

- **Tasks are instantiated and executed through service bindings.**

  In order for a task to execute, it has to use the physical services in an environment. As noted before, a physical service is a software executable available in an environment that delivers the functionality represented by the virtual service. For example, Microsoft Word can be bound to spell checking and document editing. Note that this implies that Word can also be bound to document display and character input since they are the parents in the VS hierarchy (cf. Figure 1). A physical service must provide all the interfaces (functionality, configuration and state) required by the virtual service that it binds to. This can be achieved by building these interfaces into new services or building wrappers for legacy applications.

- **A master-slave relationship is used to support fast switch over and automatic synchronization.**

  In many cases, there may exist multiple physical services in an environment that deliver the same functionality. For example, a cell phone and a PC can both provide a phone book service. A user may decide to use one service at time T and another service at time T+1 for two reasons: First is convenience: a user should be able to use whatever device convenient (in terms of location, portability, etc.). For example, a user might choose to update his phone book on a PC when he gets email, but use the phone number memory on his cell phone when he makes a call. Second is failure: a device may become inaccessible because of device failure or network failure. In order for the task to resume, the system must be able to switch over to another set of services. Given these two reasons, the system should support fast service switch over by synchronizing similar services.

Fast service switch over can be achieved using a master-slave binding. Multiple physical services can be bound to a single virtual service. When multiple physical services bind to the same virtual service, one of them is designated the master. Others are slaves. The master has two responsibilities: first, to deliver the virtual service functionality; second, to update the state of the virtual service. Similar to a warm standby, a slave service should synchronize itself with the virtual service periodically.

5.3 Task Instantiation

As outlined above, a task is a top-level virtual service that may be decomposed into a set of cooperating virtual services. A task can only be executed after its constituent virtual services are bound to physical services using a protocol called the Task Instantiation Protocol (TIP, as mentioned earlier). TIP is also responsible for configuring and initializing the services. TIP consists of three basic steps.

1. Collect information about the available physical services using external mechanisms.

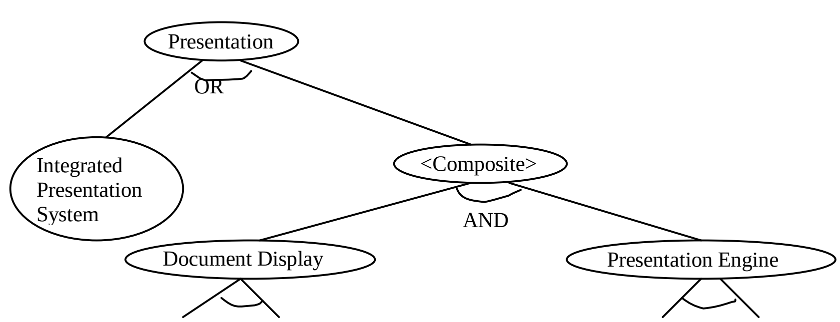
First, the system needs to find out what kinds of physical services exist in the environment using some external service registration and location mechanism such as Jini.

2. Use the virtual service type tree to bind and compose physical services.

The basic operation of this step is to check if a given physical service can bind to a given virtual service. In existing brokering systems such as CORBA and COM, to find out if a service supports certain interfaces, a broker can either check the interface signature of that service or use a run-time query. Similar mechanisms can be used to check if a physical service can bind to a virtual service.

The goal of this step is to bind all virtual services in a task description to physical services. Taking into consideration that a virtual service can be decomposed into a set of other virtual services or realized by one or more specializations of itself, this problem amounts to searching through a service configuration space.

The service configuration space can be viewed as an AND/OR acyclic graph in which the nodes are virtual services, specializations form OR links, and decompositions form AND links. Tasks are instantiated by a tree search until a path can be created from a collection of bindable virtual services to the root of the tree (the top level task). For example:



A variety of search algorithms could be used, ranging from simple algorithms such as Depth-First or Width-First search to smart ones such as those used in AI. What the tradeoffs are between different search algorithms remains an open research question.

As noted earlier, a composition script can be used to specify what services should be composed to form a high level task. However, sometimes services can't be composed together directly as specified because of interface mismatches. Ideally, in these cases, the system would automatically insert adapters to enable composition. For example, many data type mismatches can be resolved by inserting transcoders automatically (based on reasoning about datatypes, as done in TOM[4]). What kinds of service compositions can be done automatically is part of ongoing research.

3. **Configure and initialize physical services.**

After a task is instantiated through bindings and compositions, the physical services involved need to be configured and initialized. This is achieved using network-stored task configurations and state information to set the state interface of each physical service.

5.4 Achieving Proactivity

As noted before, a goal of the system is to proactively help users accomplish tasks when the direct instantiation of a task fails. At least three novel proactive mechanisms can be used in a task-drive approach.

- **Pending Tasks**

One reason that a task may not be instantiable at a particular moment is that some necessary service, device, or data is not available. This can also happen when the services do exist but can't operate successfully due to the lack of resources. The system should remember what tasks are blocking on what resources and notify a user when a task can be resumed.

The basic approach here is for the system to maintain a list of all pending tasks and their expectations on resources. Explicit support for pending tasks is critical in a task-based environment because users have no direct knowledge of low-level resource changes. Therefore, the system should prompt when a task can be done. Triggers can be installed in the environment so the system can notify a user (using a native notification service) when a task can be resumed. Triggers can be turned on and off based on user hints, such as time-to-live or other heuristics. For example, a user might specify to not be interrupted when in a meeting.

- **Task Approximation**

In many situations, when a task can't be executed, a similar one might also be acceptable. For example, when a presentation service is not available, a user may be willing to use a text viewer to just read the text. Finding good task approximations is a hard problem in its own right. A simple approach that takes advantage of the virtual service representation[2] is to generate the approximation by replacing the virtual services in the task with "similar" virtual services.

Obviously some sort of distance measurement is required in order to determine the similarity between two virtual services. A simple measurement would be to calculate the number of links traversed in the virtual tree from one node to another. For example, if two virtual services are direct siblings (with a distance of 2) then they share a common parent, which probably means that they deliver similar functionality. Other distance measurements will be investigated.

Another possible method to locate service alternatives is to use service-registered alternatives. When a service binds itself to a virtual service, it can also register another service as an alternative. When the service itself can't

[2] I expect this to be compatible with multi-fidelity computation.

operate correctly, or when the service does not exist in another environment, TIP can use this knowledge to look for alternatives. For example, a remote web-based map service might register another similar web site. The rationale behind supporting service-registered alternatives is that sometimes services themselves have a better idea of what kinds of alternatives there may be.

- **Corrective Hints**

  Services need certain resources to operate. For example, a remote display service may require a certain amount of bandwidth, a VRML viewer might need raw CPU power and memory. Similarly, some services may require the presence of certain data to operate. As noted before, the typical reason that a task can't execute is that some resource required by a constituent service is not available. Sometimes there are simple things a user can do to correct the resource unavailability situation. For example, the system may tell a user to move 50 feet to the right to get a better wireless connection.

  "Corrective hints" refer to an approach in which the system offers suggestions on how to correct the resource unavailability. Providing hints at the user level is critical because when users interact with computers in terms of high level tasks, it would be strange to require them to dig into the low-level details when a task can't be executed. The important design issue here is where the system should put the knowledge of corrective hints.

  My research uses a separate mediator (an advisor) that knows how to manipulate system conditions to make certain resources available. Services register their requirements to an advisor component. When a client sends a request to the service, the client also sends a list of "freedom variables" along with the request. The request is intercepted by the advisor component, which figures out how to tweak the variables to satisfy the requirements of the services. Freedom variables might include user location, display brightness, state of the file system (the advisor might choose to load a remote file), etc.

  The advisor frees services and clients from having to worry about corrective hints. Services should not think about what clients should do to satisfy their preconditions. Similarly, clients should not think about this issue because client developers can't be expected to know what kind of resources the services require.

  Typically, advisor semantics are highly domain-dependent. For example, a power-aware advisor may have a good idea about how to save power. A network weather map advisor might know a lot about how to change the user position to get a better wireless connection. Appropriate hooks can be built into the run-time architecture, which will make it possible to dynamically insert third party advisors that encapsulate domain expertise into the system.

5.5 Lightweight Task Creation

As mentioned before, formal task definition languages are powerful but too heavyweight for ordinary users to use every day because they require users to program using a special language. It is crucial to make it easy for users to create tasks.

- **Use system snapshots for lightweight task creation.**

  As discussed earlier, it is important to make it easy for users to create tasks. System snapshots can be used as lightweight tasks. Snapshots refer to the system state at a particular moment including the active applications, their states and user personal settings such as desktop layout. There are three reasons for using snapshots as lightweight tasks. First snapshots are much easier for a user to create than other approaches (such as system-recorded macros). Second, snapshots are natural for creating the illusion of "continuity" as users move across environments. Users can simply leave one environment and have everything reinstantiated in another. Third, snapshots capture users' computing intentions: active applications represent what users are doing.

- **Use the virtual service tree to extract multiple platform-independent abstractions of system snapshots so a snapshot can be instantiated in heterogeneous environments.**

The basic challenge with using system snapshots as lightweight tasks is that unlike task specifications, system snapshots are highly platform-dependent and application-dependent. Hence it is not trivial to instantiate them across heterogeneous environments. Suppose a user has Microsoft Word and Powerpoint running on a Windows machine, the snapshot can not be easily be moved to a Linux environment if Word and Powerpoint are not available on Linux.

It is possible to extract application/platform independent task information from snapshots using the VS tree. Given a physical service, by moving up and down the VS tree, the system can automatically locate physical services in other platforms that accomplish similar goals. A variety of algorithms can be used to do this.

To illustrate, one simple algorithm might look like this: After a system snapshot is created (using TSMP, as described in the following sections), abstractions of the state are created by walking up the VS type tree to extract the state fragments.

In the example above, after a snapshot is created, the system can realize that the task is essentially a word-editing and a presentation application using a virtual service tree. Now suppose we move to a Linux machine, the system can look for text editors and other presentation engines as physical services that accomplish similar goals.

5.6 Task State Management

The state of a task includes the state of its constituent virtual services. TSMP is used to capture task states and propagate them so tasks can be quickly and reliably transferred from one environment to another.

The key features of this protocol are:

- **Coordinate services to achieve a consistent task state snapshot.**

  Physical services must support the state interfaces specified by the virtual services they bind to. This means that a physical service should be able to export its state in the form of schemas upon request. Since a task session may involve multiple communicating physical services, distributed checkpointing algorithms will be used to ensure that a consistent task state can be achieved.

- **Use distributed task state replication to improve performance.**

  The state of a virtual service can be replicated in the following cases:

  First, task states can be replicated between master and slave services. As mentioned before, a slave service needs to synchronize itself with the master service in order to support hot fail-over or disconnected operations (elaborated in 5.7).

  Second, when a physical service is bound to a virtual service, it is possible that the physical service still has part of the task state in its cache from the last use. Incremental cache updating may be much more efficient in this case. Since virtual service state interfaces are based on structured schemas, it is relatively easy to support incremental update. For example, a timestamp can be associated with each property to remember the time of last update. Using timestamp comparison, the system can only propagate those state fragments that were changed (relative to the copy in the cache of the physical service).

- **Further enhance performance using “PUSH” technology and proxies.**

  Besides using replication, caches and incremental updates to enhance performance, it is possible to push task states proactively in the case where the system can anticipate an end user’s movement (through explicit instructions or implicit reasoning on user calendar schedule, for example). When a user leaves an environment, the system can push all the state and necessary data into the next environment that the user is going to be in. Then when a user arrives, the necessary data will already be local. Automatic reasoning about user location may require specific domain knowledge, which is beyond the scope of task-driven computing research. Appropriate hooks can be built into the run-time system so that external reasoning mechanisms can be plugged into the system. Recent work on proxies can also be used to further improve system performance (Proxies are part of the environment resource that is discovered and configured by the system.)

5.7 Service Coordination

The Service COOrdination Protocol (SCOOP) protocol is responsible for session management and maintaining task stability in the presence of changing resources, such as when services become available or unavailable. Two basic aspects of session management need to be addressed.

5.7.1 Service Plug & Play

This sub-protocol runs when a service becomes available to join a task session (either by explicit request from a user or based on preset user preference rules).

When a new service becomes available, first, the system needs to decide which task(s) the new service might contribute to. A user can tell the system how to use the new service by specifying a new binding between the new service and one of the virtual services in a task. The system can also make a decision itself using some predefined rules. Second, the new service is synchronized and coordinated with existing services, so tasks can continue with no user interference. Existing services should readjust the bindings appropriately, and the new service should be initialized. Service plug and play involves much more than existing industries work on device plug and play (such as Jini™ and UPnP™). With service plug and play, not only can a service be located automatically, it can also be automatically configured, initialized, and joined to a current running task.

5.7.2 Service Unplug & Play

A set of services may become unavailable in two situations. First, the services may not be available because of low-level resource changes, such as network failures. Second, when a user leaves an environment, he may decide only to take a subset of the devices with him. The remaining services in the environment are not available anymore.

In either case, the session is broken into two partitions. The partition the user is going to keep working in is called the active partition. Service unplug and play protocol makes it possible for users to keep working on the active partition.

When a partition is created, the state of all services in the non-active partition will be collected. The system will then reconfigure the task using the services in the active partition. If a task can't be instantiated in the active partition, the system stores it in the network and queues it until a user moves into an environment in which the task can be instantiated. Part of ongoing research also investigates how to leverage existing work on disconnected operations.

6 Implementation Issues

In order to demonstrate that task-driven computing is practical, a prototype system needs to be built. The section describes the requirements on such a prototype.

6.1 Implementation Requirements

6.1.1 Functionality

The system should support the following capabilities:

6.1.1.1 Proactive Task Instantiation

The system should demonstrate automatic instantiation of tasks on a variety of platforms (both standalone and in combination). It should also demonstrate proactivity when tasks can't be instantiated directly.

6.1.1.2 Task Mobility

The system should demonstrate that a task can be suspended in one environment and restarted in a variety of platforms. Besides demonstrating mobility of tasks, it should also demonstrate that system snapshots can be moved from one environment to another.

6.1.1.3 Task Adaptation

As devices come and go, the system should automatically adjust service configurations so that tasks can continue with a minimal amount of user disruption. Task adaptation experiments build on the task instantiation experiments. To demonstrate the scenario of new devices, after a task is instantiated and the system is told that a device is accessible and indicates the network address of the device, the system should adjust the appropriate service configurations. Second, to demonstrate the scenario of removing devices, the system should support automatic reconfiguration with both user-initiated removal and environment-initiated removal of services (for example, in the case where a device becomes unavailable because of network failures).

6.1.1.4 Platform/Service Generality

Since task-based computing is a general paradigm that can be applied in most computing systems, the above functionality must be shown to be achievable in a general setting. By generality, I mean it works on (a) most platforms, (b) can use most physical services, and (c) can accommodate most daily computing tasks.

(a) Platforms: Windows. Linux, Windows CE

(b) Services:

Display Services
    X
    Windows Display Server
    WinCE Display Server
    Structured output
Input Services:
    Desktop keyboard, mouse
    Structured input
    PDA, pen input, remote small input device
Voice based IO Services
Various kinds of document editors
Various kinds of image editors
Media processing:
    Audio input services
	Instant Messaging: Windows Popup, Zephyr
	Email: various POP/IMAP clients
	Data transport:
	    FTP, HTTP clients
	    DFS drivers
	Type transcoders: TOM, various document converters on UNIX, image distillers
	PIM Services:
	    Address book
	    Calendars
	    Database & directory services
	Web-based services such as MapQuest

Table 1. Physical Services

(c) Tasks:

It is crucial to make sure that the techniques developed can support a representative set of tasks. Task-driven computing can be demonstrated using a subset of these tasks.

- Complex document viewing and editing that involves embedded media clips and spreadsheet
- Making Presentation
- Contacting a colleague
- Message Notification, for example letting the user know when a message about security arrives
- Driving direction & map request (using a web based map service and text distiller with varying fidelity)
- Annotation (annotating some information using available services, possibly handwriting)
- Reminders to do something
- Meeting scheduling

These tasks are chosen for the following reasons. First, they are the typical day-to-day tasks. Second, they represent enough coverage to demonstrate the protocols. All of them can be instantiated into dynamic service coalitions on heterogeneous platforms. They all can be brought from one environment into another. Third, they represent coverage of a variety of physical services that they can take advantage of.

6.1.2 Usability

In order for the system to be truly useful, it has to be easy to use. In other words, the system should reduce user attention demands and demonstrate acceptable performance.

6.1.2.1 Reduction of User Attention Disruption

The amount of user attention disruption can be measured in terms of the amount of time and effort a user spends on dealing with non-task related work such as filling out things in option dialogs, starting and closing applications, and converting files. A prototype system supporting task-driven computing should show that it dramatically reduces the non-task related working time.

To measure user attention disruptions, a small agent can be developed on each of the platforms to monitor and log user actions such as mouse clicks, dialog boxes and typing. The logs can be analyzed to figure out how much time a user spends on non-task related efforts. To make the measurements more accurate, the agent can let a user indicate when he begins to work on a task and when he stops.

6.1.2.2 Adequate Performance

The system should automatically instantiate tasks with acceptable delay between the moment a user sends a request and the moment the task can be resumed in different environments. It should also adjust the physical service configuration with an acceptable delay between the moment a service changes its availability and the moment a task can be resumed.

Performance of system reconfiguration and task instantiation can also be measured using the agents mentioned above. As for the measurement the performance optimization strategies mentioned before, the features can be enabled or disabled in the run-time components. Measurements can be taken for comparison.

6.2 Current Status

I have built two prototypes to demonstrate the three protocols. The first prototype supports multi-device enabled software services, and demonstrates a simple version of the SCOOP by allowing client devices to dynamically join and leave a session. The second prototype, DeskSpace, allows users to create tasks (on Windows) by taking system snapshots and then move the tasks to another PC. It demonstrates a primitive version of TSMP and TIP (in Windows environments).

7 Conclusions

This paper describes the concept of task-driven computing, its motivation and the technical approaches that can be used to address the challenges it poses. By enabling users to interact with computer systems in terms of high-level tasks without bothering with low-level service configuration details, task-driven computing is an innovative way to reduce the attention and knowledge requirements of mobile users in a pervasive computing environment.

8 Acknowledgement

This research was sponsored in part by the Defense Advanced Research Projects Agency under agreement numbers F30602-97-2-0031 and N66001-99-2-8918. The U.S. Government is authorized to reproduce and distribute reprints for Governmental purposes notwithstanding any copyright annotation thereon. The views and conclusions contained herein are those of the authors and should not be interpreted as necessarily representing the official policies or endorsements, either expressed or implied, of the Defense Advanced Research Projects Agency Rome Laboratory or the U.S. Government. The authors would also like to thank Bradley Schmerl for his comments on the draft.

9 References:

1. M.Shaw and D.Garlan, *Software Architecture*, Prentice Hall 1996
2. M.R.Barbacci, D.L.Boubleday and C.B.Weinstock, Application-level Programming, *Proceedings of the Tenth International Conference on Distributed Computing Systems*, June 1990
3. G. Leavers and M. Sitaraman, *Foundations of Component-Based Systems*, Cambridge University Press, 2000
4. J. Ockerbloom, *Mediating Among Diverse Data Formats*, PhD. Thesis. Technical Report CMU-CS-98-102, Department of Computer Science, Carnegie Mellon University
5. Brad Meyers, Herb Stiel and Robert Gargiulo, Colloboration Using Multiple PDAs Connected to a PC, *Proceedings of CSCW 98: ACM Conferences on Computer-Supported Cooperative Work*, Nov 1998
6. Jini Connection Technology, *http://www.sun.com/jini*
7. Universal Plug and Play, *http://www.upnp.org*
8. HP E-Speak, *http://www.espeak.hp.com*
9. CORBA, *http://www.corba.org*
10. Component Software Model, *http://www.microsoft.com/com*
11. Javabeans Component Model, *http://www.java.sun.com/beans/index.html*
12. M.Satyanarayanan, Mobile Information Access, IEEE Personal Communications, Vol.3, No. 1, Feb 1996
13. B. Noble, M.Satyanarayanan, et al, Agile Application-Aware Adaptation for Mobility, *Proceedings of the 16th SIGOPS*, Oct, 1997
14. P.Nixon and V.Cahill, Mobile Computing: Technologies for a Disconnected Society, *IEEE Internet Computing*, Jan 1998
15. Alan Dearle, Toward Ubiquitous Environments for Mobile Users, *IEEE Internet Computing*, Jan 1998
16. Application Configuration Access Protocol, *http://asg.web.cmu.edu/acap/*
17. J.Jing, A.Helal and A.Elmagarmid, Client-Server Computing in Mobile Environments, *ACM Computing Surveys*, Vol. 31, Number 2, June 1999
18. Sun Ray, *http://www.sun.com/sunray*
19. Workflow Management Coalition, *http://www.aiim.org/wfmc/mainframe.htm*