Software Upgrade in Ubiquitous Computing.

John J. Barton, HP Labs, Palo Alto CA.

Abstract. Adequate approaches for component upgrade should be required among the critical properties when designing systems for ubiquity. Ubiquitous computing creates new challenges for upgrade. To illustrate the challenges I begin a list of the historically successful strategies for systems of large scale deployment and their critical ingredients for success. Then I consider the impact of ubiquitous computing on these strategies and I examine the impact of upgrade on the practical viability of two proposed ubiquitous computing systems. This gives us starting point for understanding an important aspect of realistic ubiquitous computing.

1 Introduction

Software upgrade has become an annoying, continually present feature of personal computer use. In ubiquitous computing, we have more computers near our person. Each of these will need continual upgrades. Unless ubicomp systems facilitate upgrade, user will be harassed by a stream of upgrades or failures related to upgrade failure. This does not resemble of the world of ubicomp imagined by Weiser[1]. However, upgrade is not a topic covered by books on distributed computing [2]; it is not even considered a critical challenge for ubiquitous systems [3]. Upgrade failure impacts deployed systems, long after they have held the attention of systems researchers. In this paper I will make the case that, as we enter a time of "ubiquitous computing" with increased embedded, "shared" or "public" rather than "personal" computing, software upgrade must become a central issue of system design.

My goal in this paper is to put “upgrade” on the list of critical systems issue for ubiquitous computing, comparable to way "scalability" or "fault tolerance" are listed in distributed computing. These two properties are not essential for exploratory systems, but experience guides us to analyze for these properties before bringing distributed systems out of the laboratory. I will introduce the idea that similar analysis should be made for ubiquitous systems with respect to upgrade: failing to provide adequate upgrade may prevent some otherwise promising system topologies from succeeding in deployment.

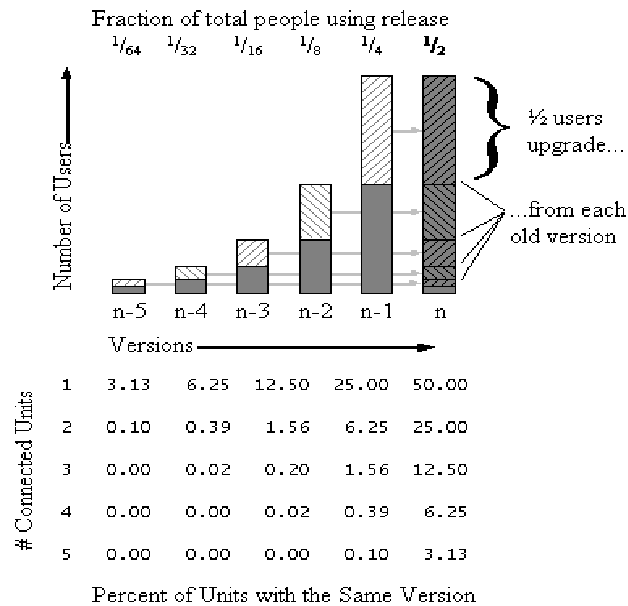
Many mechanisms for upgrade in *distributed systems* have been reported [see examples in 4]; I do not propose a new mechanism here. Indeed my central point is that one of the fundamental differences between *ubiquitous* and distributed systems is the impracticality of distributed systems notions of upgrade in ubiquitous environments. Device and task heterogeneity, mobility, and physical integration make ubicomp systems different: other approaches or combinations of approaches to upgrade are required. Consequently I will survey approaches to the problem of multiple versions in large, physically-dispersed, communications systems including but not limited to distributed systems then discuss how they nature of ubiquitous computing impacts these approaches.

In this paper an *upgrade* is a change to a component in a system that alters its interface to other components or alters the semantics of that interface. I'll call all other changes *updates*. These definitions can only be applied by the end users after a component has been changed. Thus a “bug fix” or “patch” to a component typically is intended by designers to be an update. However it can become an upgrade because of some unanticipated issue in the deployed environment of the component. This is one illustration that the challenge of upgrade in ubiquitous computing centers on the difference between the actually user-experienced system and the system imagined by the designers.

2 Why ubiquity makes upgrade critical.

To understand why upgrade is so important to ubiquitous computing systems, consider how it came to be important for personal computing. Most readers will be personal computer users and most will have personal experience with problems in upgrading them. An isolated personal computer needs upgrade only when the user finds compelling new features in available software or hardware upgrades. Individuals need only balance the value of these new features against the economic and personal inconvenience cost of the new component and the upgrade process. Once a personal computer comes in contact with other computers via networking, upgrade becomes complex: as the software on other networked computers changes, upgrade may be required to continue to communicate (or in the case of security, to prevent unwanted communications). Communications requires consistency in physical and software interfaces, protocols, and data formats. Thus communications forces upgrade on the personal computer users, forcing them to readjust their economic and convenience priorities.

To illustrate this point in another way, consider a trivial model of communicating PCs with no upgrade strategy at all, as illustrated in Fig. 1. Suppose upgrades for the system come out at regular intervals and suppose that each new version has enough new improvements and nice features to cause half of the users of each old version to upgrade to the new one. This seems reasonable since the producers of new versions will probably add features to a version until they can attract a sizable fraction of users, but not delay more than necessary. We ignore the issue of new users or users who quit using the overall system. This too is reasonable, if we are assuming that the system is “ubiquitous” so that everyone wants some version of it. Thus, by design, the primary unreasonable aspect of this model is the absence of an upgrade solution.



# Connected Units					
1	3.13	6.25	12.50	25.00	50.00
2	0.10	0.39	1.56	6.25	25.00
3	0.00	0.02	0.20	1.56	12.50
4	0.00	0.00	0.02	0.39	6.25
5	0.00	0.00	0.00	0.10	3.13

Fig. 1 Simple model for upgrade in a ubiquitous system. Each version attracts half of the users from all previous versions, summing to half of all users. The top half of the figure illustrates the relative number of users for each version. The bottom half is a table of the integer powers of the fractions in each version, giving the percentage of common versions in a random collection of connected computers.

Despite users desire for the system, upgrade is still costly, in money and lost productivity. Obviously if only half of the users upgrade, as shown at the top of Fig. 1, half of all users will hold back in a spectrum of older versions. This effect is well-known to Web site builders. The evolution of web-browsers creates a spectrum of clients; site builders are advised to avoid using features of the latest clients if ubiquitous readability is desired[5]. This "upgrade friction" is an intrinsic property of peoples variable evaluation of the cost/benefit ratio for upgrades.

Consider the impact of attempts by this community of users to communicate with each other. As shown in the bottom of Fig. 1, if new versions of a system break the communications between computers, multiple-computer communication becomes almost impossible. If you have a computer with the most recent version of the system, you'll only have a 25% chance of contacting a computer you will be able to work with. If you have a version only one generation out of date and you want to connect to two other computers, you will only have a 1.5% chance of success. Clearly upgrade in a system of communicating computers must have some strategy for avoiding such failures.

Communications changes an isolated personal computer into an element of a larger system, placing new demands on components that the original upgrade strategy may not have anticipated. Essentially similar effects have happened when originally monolithic applications were re-architect ed into modular systems built on reusable libraries: the re-use added programmer productivity but took away end-user productivity during upgrades; indeed, new products have begun to appear that reverse this historic pattern of re-use by bundling applications in to completely standalone units[6].

The restructuring personal systems into subsystems in a communications network amplifies when we turn to ubiquitous computing. The combination of prevalent mobile personal computers and wide-spread access to the Internet causes "personal computing" to resemble "ubiquitous computing". Specifically, being able to use the full capability of one's mobile PC in new physical environments requires some support for "physical interaction" and "spontaneous interoperation", systems problems claimed by ubiquitous computing research[3]. Therefore, the solving the networked PC upgrade problem will not suffice, we need an approach to upgrade for ubiquitous computing.

In what sense is ubicomp upgrade different then? A key difference is perspective: the other systems are described from the system designer's point of view while a ubiquitous system is what the user experiences. In ubiquitous systems, people interact with *multiple computers in a physical environmen*t (physical integration [7]). Each computer could be part of different distributed systems, but now they interact with each other. For example, a personal computer running Windows might work with a personal video recorder operated on the Tivo network.

In addition ubiquitous computing postulates people carrying computers *across administrative boundaries* expecting these computers to interact with computing resources in all environments they encounter (spontaneous interoperation [7]). For example, users carrying cellphones and laptops move their machines between work and home environments. Mobile computers combined with physically integrated computers means computers from multiple disjoint administrative domains communicate. Moreover, ubicomp seeks spontaneous interoperation which *limits the user's time* that can be side-tracked for upgrade.

The issues that arise in these cases are often considered to be "user-errors" from the perspective of a distributed system: laptop users *should not* be using work machines for games at home; users *should not* be installing unconventional software; backlevel versions *should not* be relied upon.. But an underlying premise in ubiquitous computing is the "computerization and interconnection of everything". As more computing objects become embedded in our lives, we will have more inter-domain and context-dependent upgrade issues. We are moving from a world of "should not" to one of "always will": can we develop approaches to upgrade that succeed in real ubiquitous systems?

As a first step Sec. 3 discusses strategies from four widely-deployed systems: telephone systems, distributed systems, Internet infrastructure, and online computer games. Ubiquitous computing overlaps all of these cases: we talk about systems of coordinated computers[8], systems with telephone-like simplicity[9, 10], systems based on Internet technology[11], and community-focus systems like online computer games[10,12]. Sec. 4 will examine how ubiquity impacts these strategies and how some of these systems will be impacted by an upgrade requirement.

3 Upgrade Strategies.

Having made the case that ubiquity makes upgrade more common and more difficult, raising its importance in ubicomp system design, we next examine successful and unsuccessful strategies for dealing with upgrade in existing systems that share only some of the characteristics associated with ubicomp systems. Table 1 summarizes the strategies I discuss here.

Strategy	Example	Prerequisites
Intermediation	Telephone to SMS-enabled cell phone	Legacy infra-structure; standard format.
Administered	Corporate distributed system upgrade.	Administrative control; adequate knowledge of endpoints
Multiprotocol	HTTP 1.0 to HTTP 1.1	Version discovery; layered protocol
Online	Multiplayer Internet Computer Games	Uniform client; download bandwidth

*Table 1. **Summary of Strategies for Upgrade**. The strategy column refers to the subsections 3.1-3.4. The example column gives illustrative successful examples uses of the strategy. The prerequisites column lists properties need for the strategy to succeed.*

As with most aspects of system design, the issues in real systems are complex. Consequently I look for some general principles and reason by examples that have “succeeded” and “failed” or perhaps failed to succeed as much as expected. The two principles I want to focus on here are 1) the interaction of system topology and upgrade, and 2) critical prerequisites for success with a given upgrade strategy. This will set the stage for Sec. 4 which looks at proposed ubiquitous system topologies and at how ubiquity impacts the prerequisites for upgrade strategies.

Many readers will have personal experience with an important systemic problem of upgrade: upgrade sometimes breaks components seemingly unrelated to the upgrade. This problem is a consequence of the interdependency of components in a system “open” [2] to extension. For inconvenience's in the following discussion I'll use two new terms for aspects of this problem. A *ripple event* occurs when upgrade of dependent component forces upgrade on an antecedent component. An o*rphaned branch* is a dependent component that relies on an older version of the antecedent component which is altered, removed, or obscured during a ripple event. For example, both a word processing program and an Web browser might rely on a socket library for networking. Upgrading the Web browser might force the socket library to be upgraded, rendering the word processing program unusable. Note that the designer of the Web browser upgrade may be unaware of the existence of the word processing application and vice versa. This incomplete communications between designers is built into the nature of an open system. To say this another way, upgrade creates a force that favors closed systems under a single designers control. Unless one chooses

to imagine a future with a single global software system, ubiquitous systems must solve the upgrade problem to remain open to extension.

Both operating systems and distributed systems would consider orphaned branch components to be erroneous. These disciplines strive for a consistent state. At least in my personal experience, users consider formerly functional but orphaned components to be “correct” and in their view the problem lies with the upgrade and its ripple event. That is another illustration of the principle that a ubiquitous system is the one experienced by a user, not necessarily the system designed by an architect.

3.1 Intermediation Strategy.

Intermediation is a strategy to achieve backward compatibility between old and new technologies without discarding the old technology. It goes by other names, including content adaptation and transcoding when applied to Web networks. The idea is to place a translating component between new versions and old versions of components that must communicate. Intermediation can also be used between two competing components of a similar generation.

Referring to the model in Fig. 1, intermediation works to allow version (n-1) to continue to communicate with version (n) by introducing a third component designed to translate between them.

This strategy is commonly applied in communications systems that use an "infrastructure" network. By this I mean that nodes connect to a network, then address remote nodes through infrastructure the network provides. For example, land-line phones connect to relay stations in the phone system. The infrastructure provides critical value in scaling the system to large numbers of nodes: multiple conversations can be multiplexed over the common infrastructure, reducing the per-node cost of the system. But the infrastructure represents a significant system cost that eventually becomes a “legacy” infrastructure. Sub-networks with new capabilities can be added to the network via "gateways" that adapt communications protocols between upgraded and back-level nodes. Thus wireless phones connect to land-line phones through gateways that adapt radio-waves to voltages waves. The cost of extra components to perform intermediation at the edge between new and old networks can be borne by the infrastructure provider and amortized across multiple users.

To be useful, intermediation requires valuable legacy; to be successful it needs a translatable data format. For phones the valuable legacy is the installed land-line phone wires and the data model is analog sound vibrations. Gateways in the phone system transcode these vibrations into myriad encoding for different parts of the complete network. Data falling outside of this common set, for example SMS messages, does not pass into older parts of the network since the clients there cannot use this data. Eventually intermediation may be replaced by new systems.

As a strategy, intermediation is immune to ripple events but provides lower function than solutions that propagate changes. That is, since intermediation by design insulates downstream nodes from changes in upstream nodes, the communications is limited to the lowest common denominator data format. Consequently, intermediation combined with a carefully selected data format leads to a robust communications system. This is another form of the “end to end” argument[13]: by passing uniform, content-agnostic packets, Internet routers and their end points can be upgraded in

many ways with minimal impact on the system as a whole. For example, IP packets can flow over Ethernet or fiber optics without altering the web-browser applications that use IP packets.

I described intermediation from the perspective of a network. From the perspective of a user having a new kind of network node, intermediation looks like 'necessary evil' rather than an upgrade strategy. A USB-serial converter, analog audio from a CDROM player, JNI in a Java development environment: for new versions with new technologies to work within an existing system, some otherwise extraneous adapter is typically needed. These adapters add cost and complexity, working against the value of the upgrade. Thus a significant design issue in intermediation is the variety and number of back-level versions that a new component must interface with. Similarly, we need to choose where the intermediation is placed. When intermediation is embedded in an network (like cell-phone towers) the initial cost of deployment is high, but per node cost are lower.

3.2 Administered Upgrade Strategy.

A second strategy for upgrade applies to distributed systems controlled by a central administrator [4,14]. In this strategy a central service actively pushes communications endpoints forward to new versions. For example, all of the routers in a local area network could be upgraded on a weekend by an installation crew hired by a corporation. After the upgrade there are no back-level clients: all of the cost of upgrade is incurred at once.

Referring to Fig. 1, administered upgrade tends to push a collection machines towards a single version, usually one less than the most recent available version (n-1). The conservative choice improves upgrade success because typically the most recent version of a component has some bugs that do not appear until it is used by large number of people.

Two ingredients seem essential for administered upgrade. First, one administrative authority must have access to all of the end points. This implies a boundary outside of which we have no need of interoperation. Second, systemic consequences of the upgrade need to be predictable. For example, new routers in a LAN upgrade need to operate with all of the existing computers on Monday morning.

The administrative authority requirement is easier to fulfill with some system architectures than others. A centralized, immobile system with lightweight (“dumb”) terminals, like a mainframe, computing cluster, or conventional phone system, will likely have both authority and a clear model of the central core. When upgrade is need on remote nodes—say SMS messaging support for telephones---the challenge increases. Systems with more function on remote nodes have more difficulty: the Internet just cannot use administered upgrade.

The predictability requirement also interacts with system architecture. A complex system makes predicting the systemic impact of an upgrade too difficult even for experts[15]. Upgrade mechanisms can reduce these problems to some degree. Asynchronous upgrades help reduce the aggregate impact of upgrade failure and rollback strategies make upgrade more predictable[14]. However, success in administered upgrade will always be limited by the continual changes in hardware and the variety of uses for general purpose computers.

Commonly experienced failures of ad mistered upgrade result from breakdown of the administrative domain. As corporate computing services have moved to laptop computers that can operate out of reach of any central administrator and that can be used for purposes other that the ones envisioned by the administrators, the illusion of a "distributed system" gives way to the reality of a set of autonomous communicating systems with some common components. When laptops users install software that depends on the software maintained by the corporate system, the administered corporate upgrades trigger ripple events, orphaning components installed by end users. While these components might not be sanctioned by the corporation, the end user experience is certainly poor.

3.3 Multiprotocol Strategy.

In the multiprotocol strategy, some or all of the endpoints offer communications via both back-level and upgraded protocols. For example, a typical web-server will support multiple client protocols, including different versions of HTTP and HTML languages. As a consequence of multiprotocol servers, the system as a whole can support both back-level and upgraded clients, allowing a graceful transition during which users learn about the added value of the upgrade and consequently seek to install it.

Referring to Fig. 1, the multiprotocol strategy assumes that version (n) through (n-i) must continue to communicate, even if supporting this backwards compatibility costs resources that would not be needed for communication between components running just version (n).

The multiprotocol strategy can be implemented in two different ways. First a new protocol can be "backwards compatible", meaning that nodes using the new protocol can communicate with back level nodes by design of the protocol. This is typically implemented as a *multiprotocol server*. For example, HTML 3.2 can be read by any client capable of reading HTML 3.0 even if some added features will not appear in the way they will on an HTML 3.2 client. Second, the new protocol can be implemented in parallel in upgraded nodes ("*dual stack*"). For example, a computer can run both an IP v4 and IP v6 networking stack simultaneously.

Two ingredients seem critical for this upgrade strategy. First there needs to be some way for one side of a communication to discover the version of other side. In web-server example, this comes from the client-server architecture (so the version of the client matters only to the server) and from the HTTP header structure that includes information about the client capabilities. This discovery process builds in latency and additional points of failure in to protocol, creating a design tradeoff.

Second, the communications protocol needs to be layered to decouple changes. Changes in communications interfaces between two layers can be insulated from and independent of other layers. Thus in a web-server the version of HTTP determines the transfer of HTML but not what version of HTML will be transferred. The thinner the layer, the less software will be impacted by a protocol change; the thinner the layer the more layers are required to build up function. Historically this design tradeoff has shifted to thinner layers aided by the synergistic effect that thinner layers are easier to standardize, code, and debug.

Some kinds of systems have difficulty exploiting a multiprotocol server upgrade strategy. Before the wide-spread adoption of HTTP technology, systems based on Remote Procedure Call (RPC) were considered to be the foundation for developing interactions between remote systems. Despite an enormous investment in technology, RPC systems and the Remote Method Invocation (RMI) systems that followed did not fulfill their original expectations. While the RPC technology fits the programmer's needs in activating functions on remote machines, the procedure call approach create communications protocols with deep, tightly coupled interactions. This compromises the system's need for an upgrade strategy. (see also [16]).

The multiprotocol strategy minimizes ripple events by building intermediation in to the upgrade process. In effect each upgrade carries with it intermediation with previous generations. Thus, like stand-alone intermediation, the multiprotocol strategy gains stability by isolating changes to pairs of components (layers of a protocol stack in this case). This also means until protocol changes are propagated to other layers a communications stack or to other nodes in the network, the multiprotocol strategy leads to a low-common denominator communications network. Unlike stand-alone intermediation, multiprotocol solutions can trade upgrade propagation risks against improved communications simply by controlling how many components are upgraded at a time.

3.4 Online Upgrade Strategy.

A fourth strategy for upgrade uses the communications system itself to install expanded capability into back-level nodes. We should distinguish online upgrade built into a system from co-incident use of the system for its upgrade. So if a user downloads a new web browser, the communications system is being used to upgrade the system, but this is roughly similar to the case of a user who buys a new version of the browser on CDROM. Online upgrade becomes more of a system strategy when the system directly encourages or automates this procedure. For example, many networked multiplayer computer games redirect users to servers with upgrades when they attempt to connect with a back-level client[17]. Online upgrade has become a central element in the user interface of the most recent version of Windows [18].

Referring to Fig. 1, online upgrade tries to lower the barrier for all users to move to the latest version. In some sense it is a "pull" version of administered upgrade's "push" model. End points that will be upgraded request the upgrade rather than having a central authority distribute the upgrade. In practice these strategies act differently because of the different expectations that upgrade designers have for the system architecture. Typically administered upgrade seeks to make all machines in a domain of control run "the same" system; online upgrade usually operates on a single application. Thus administered upgrade considers variations in systems to be problems while online upgrade expects and guards against variations. Nevertheless online upgrades are typically less ambitious. They work within a context, say an operating system and application suite, that it expects to be version consistent.

Online upgrade relies on two critical resources: ubiquity of the underlying client environment and a communications path with enough bandwidth to download the update in a usable time. For example, online computer games often rely on Microsoft operating systems for graphics and network access; many online computer game

players access the Internet with bandwidth beyond those supported by telephone modems[17]. The resources of ubiquity and bandwidth are interrelated in that when an existing ubiquitous client has more features useful for the communicating system, smaller downloads will be needed. HTML forms and Java Applets are one extreme where the client has a great deal of base function and only a small set of instructions are needed to "upgrade" the client. In these cases the upgrade is so small that we don't even bother to save it on the client except possibly via client caching.

When an upgrade size reaches the limitations of bandwidth, the online upgrade strategy can fail. Even with highly motivated users of online games, large upgrades can result in lost users [17]. When ubiquity breaks down, the strategy can also fail. For example, Dynamic HTML (DHTML) was promoted by Microsoft as an upgrade for HTML, but it required specific versions of one browser, Internet Explorer 4 or larger. The servers offering DHTML did not have enough clients to justify their deployment [19]. By contrast, when Macromedia developed Flash they built a software layer for the downloaded content that runs on many different browsers on different operating systems. By this means they insured that Flash-based applications would have a ubiquitous client environment. Of course the features of DHTML and of Flash differ and these differences may account for their relative success, but at least some analysis suggests that upgrade strategy was critical [19].

4 Application to Ubiquitous Computing.

Having made the case for the critical role of upgrade in ubiquitous computing and introduced some strategies based on historical successes, we can discuss how they may apply to ubiquitous computing. Then I give a couple of examples primarily to introduce the idea that systems designers should include upgrade analysis as part of the set of questions they use in evaluations. Here is the critical point: in designing systems for the new constraints of ubiquitous computing, a promising approach in the lab may fail in deployment unless upgrade is considered.

4.1 Impact of ubiquity on upgrade strategies; suggestions for improvements.

Let us examine the strategies of Sec. 3 when applied in systems of multiple, specialized (non-PC, appliance-like) computers under multiple administrative domains, operated in a “spontaneous fashion”, where stopping to apply an upgrade is not welcome.

Intermediation requires a common data format to translate between components. With multiple computers the probability of finding a common format goes down; the translation functions available under one administrative domain may not be available elsewhere; installing the translation manually interrupts or prevents successful spontaneous uses. Thus widely-deployed, broadly-applicable, non-proprietary standards for data formats are vital for ubiquitous computing.

To the extent that the intermediation can be accomplished on a central node or server, we avoid many of the problems of intermediation in a ubiquitous system. This suggests the introduction of “intermediation servers” like the transcoding systems

[20] in ubiquitous computing environments. This might alter the system topology: a peer-to-peer system with local communications may operate like a client/server system if every peer is in fact communicating with an intermediation server.

Administered upgrade requires authority and system understanding. In many ubiquitous systems, multiple mobile computers interact with fixed computers: the authority to upgrade is split between the mobile user and the facility with the fixed computers. Therefore administered upgrade can not apply to the overall system unless all participants cede upgrade authority to a common point. Something close to this is happening in the PC market where Window's (online) update function increasingly replaces independent organization's upgrade systems. Administered upgrade discourages extension and "openness" except within the authorized upgrade tree: any extension outside of that tree have a chance to be orphaned. Users of corporate or institutional upgrade systems will recognize this effect: the official software suite is the only one that gets supported by upgrades over time.

To the extent that ubicomp systems rely on "appliance" like or "utility" computing, a system of multiple computers could be upgraded by an "oligarchy" of global administrators, each familiar with a particular special purpose machine and with the upgrade plans of the other administrators. This already matches the industrial practice in many areas of "standardization" where teams from different companies meet to agree on device interaction protocols. Extending this practice to upgrade would provide the system understanding needed to support wider application of administrated upgrade.

Multiprotocol strategy requires version discovery and layered protocols. Most communications protocols include some version discovery or negotiations step. However, in a ubicomp system, the heterogeneity of devices and diversity of environments they operate in creates a "meta" discovery problem: before we can even compare versions of a protocol we need to find out what kinds of communications channels we share, or what combination of networks we can use to connect. Similarly the end-use focus of ubicomp requires layering all the way up to the elements of the system the user experiences. We currently have an excellent stack up through file transfer, but the files we transfer tend to have data types specific to versions of applications.

Work on vertical handoff and especially on connection diversity[21] provides the first steps towards expanding the multiprotocol protocol strategy for ubicomp. The introduction of XML is an example of a new layer for parsing documents. The next steps for ubicomp information exchange needs to focus on application and version independent semantics. An important property of Web client/server system also needs to be exploited in ubicomp stacks: skeptical software. When a web client receives a web page it attempts to display it even if the text is not in the right format or markup tags not understood by the client are present. This property allows backlevel clients to work with new servers, a kind of generic multiprotocol strategy.

On-line upgrade requires bandwidth and a similarity in environment. Disconnected operation and the use of low power devices works against online upgrade by limiting bandwidth. But the most significant problem is environmental diversity: as a diversity of sensors or networks are added to a machine it becomes more custom and less likely to upgrade without errors.

Coupling upgrade with power charging is already a feature of PDA devices developed for use with PC operating systems. This suggests a ubicomp "re-charger"

that features nightly upgrades for other mobile devices and it suggests building network access into fixed ubicomp devices for online upgrade (as already has been done in the products from Tivo for example). Environmental diversity can be minimized by systems that treat sensors like external machines are treated: communicate with them using multiprotocol stacks or intermediation. This allows the core machine to remain “stock” and able to upgrade from online source gracefully. As long as sensors are handled by device drivers built into the operating system, the pressure from the (failure of) upgrades will favor systems of homogeneous sensors with homogeneous function. Autonomous sensors or sensors communicating via common ubiquitous data formats[11] will be more successful in deployment.

4.2 Upgrade and System Topology.

The other way we can use an understanding of the importance of upgrade is in the analysis of new proposed systems. To give a flavor of how this can work I will pick on two examples from the ubicomp literature, mobile devices working together and mobile devices working with fixed devices.

4.2.1 Ad hoc routing. Peer-to-Peer mobile devices.

A large number of new Internet routing protocols have been proposed under the general heading of "ad hoc routing" [22]. These protocols feature communications without requiring "infrastructure". For example these systems seek to allow a collection of mobile users of computers to communicate over wireless links with IP packets but without nearby access points relaying the packets over wired networks.

Design for upgrade in an ad-hoc routing network will be critical: a glance at Fig.1 shows that any collection of multiple machines will be unlike to run the same version of a component. Without infrastructure, a back-level and an upgraded node will be unable to use intermediation to communicate and since they are not "online", online-upgrade is not an option either. Since the topology of the network at the IP layer is not client-server, a simple version of the multiprotocol server strategy does not help.

This leaves us with multiprotocol upgrade on each node. However as more nodes join the network, only features in the lowest common denominator protocol will be transmitted. If the protocol is structured to maximize the common data model, then some amount of intermediation can be built into the protocol, that is new nodes can up-convert packets for backlevel nodes. If every node was multiprotocol (so they can communicate) and every node could upgrade other nodes (so after communications they can upgrade), then upgrades could propagate through the network at the cost of bandwidth, power (on mobile devices), and user trust. These speculations beg study: any upgrade strategy would impact the performance and energy analysis the network; it seems like subject worth further research.

"Application layer" P2P systems [10,12] have similar issues. These systems leverage local area networking between mobile devices to provide a new communications/computing model. Following the pattern set up by Internet application models, they already use protocol layering. However, unlike many Internet application models, these mobile P2P applications don't rely on client-server

topology: multiple peers are expected to interact in each session. Therefore these applications would resemble ad-hoc networking in update, except that the chances of a common data model needed for intermediation seems lower because the applications are more complex and specialized, leading to more complex data exchanges. Note that the P2P file sharing applications on the Internet use the combined multiprotocol/online-update strategy to propagate updates [23], and approach that would not be available in purely ad-hoc mobile systems. Again the overall performance and usability of such systems must consider the impact of any update strategy.

4.2.2 Personal Server and Public Access Displays.

As a second example of a ubiquitous computing proposal, consider a "pervasive" system of publicly accessible input/output devices. Conceptually, pervasive I/O devices mean users no longer need to carry keyboards and displays to use computing systems. For example, the Personal Server [24] project proposes a tiny server with only wireless networking for user interactions. A user approaches a public display or printer and uses its keyboard to pull information out of the Personal Server.

In one perspective, the Personal Server project re-aggregates the components in a personal computer. The wireless network interface card and the personal disk drive combine to become the Personal Server. The keyboard, mouse and display become the public access display. Re-uniting these components leads to user-experience, communications, and physical platform challenges [24], but to be successful the system of personal servers and public devices need to communicate in the face of updates on either side. For this purpose, we need to consider the public I/O devices to be "servers" in that they are in position to implement the update strategies for various mobile clients. Thus a public wireless access point could provide intermediation for local area printers, allowing different versions of a Personal Servers to print [25]. Or the public display could offer multiple protocols for connection to different versions of the Personal Server that might wish to connect. Finally, these public devices are likely to have access to both the Internet and electric power that would enable them to employ online update for themselves and for any Personal Servers that come in range.

Each strategy will have significant impact on the design, deployment, and cost of the public access device. An especially vexing issue for upgrade in these systems is the asymmetry of authority for upgrade. Version control on the personal server follows a personal evaluation of cost/benefit; version control on each different public device follows the cost/benefit evaluation of each different hosting facility. While this kind of issue is not commonly considered to be a systems issue, they must be resolved for successful deployment of these kinds of ubiquitous systems.

5 Contribution and Open Issues

To conclude let me review what I believe this paper contributes to ubiquitous computing. First, I argued that "upgrade" should be listed among the critical systems properties when designing for ubiquity. Second I began a list of the historically

successful strategies for similar kinds of system and the critical ingredients needed to use them. Third I examined how ubiquity impacts these strategies and suggested some consequences. Finally I applied these ideas to some proposed ubiquitous systems. Together these contributions represent a starting point for understanding an important aspect of realistic ubiquitous computing.

I described strategies for system success in the face of upgrade; they were not necessarily strategies for the system upgrade. Thus intermediation and multiprotocol servers appear to be "work-arounds" that try to keep a system successful for users. Administered upgrade and online-upgrade are more like systems solutions to the upgrade problem, but my intent was to show how they impact system design, not to advocate for or against their use. An interesting but different topic would seek a upgrade solutions for ubiquitous computing. Partly this could build on upgrade in distributed systems. Carzaniga et al [26] review a number of upgrade systems (up through 1998) and provide a detailed framework for characterizing such technologies. See also [4] for notes on upgrade systems. But ubiquitous systems are not distributed systems: new approaches like the intermediation servers, connection diversity, upgrade stations, and autonomous sensors are needed.

For further progress in understanding upgrade, the interrelated issues of security and usability need to be considered. Superficially we could consider the security implications of each strategy. From the security point of view, intermediation is the safest strategy. The introduction of upgraded clients challenges the security of the back-level clients only through any special privileges granted to the gateway. This makes security analysis and containment are relatively easy. Security issues in multiprotocol servers are also localized, in this case to the servers. The risks are higher in proportion to the number of back-level services that must be supported of a multiprotocol service. In administered upgrade, all users have to trust their administrator and the changes are spread over many machines. Any violation of the assumption that all the machines are identical can make the upgraded machine vulnerable to attack through improper upgrade. Conversely, identical machines make the system more vulnerable to viral infection of during upgrade. Finally, online-upgrade requires the user to trust both the peers that it tries to communicate with and the online upgrade server. Insuring security in such a system remains an open question. However, this last example also illustrates how complex security issues in ubicomp can be. As anyone familiar with today's Internet software explosion will realize, users often place high value on usability compared to security. Thus online upgrade is widely applied despite its dubious security properties. A useful understanding of upgrade security requires understanding how it relates to usability.

Of course the ultimate system strategy for upgrade avoids it. The layered communications stack and the standardized data model are examples of system techniques that reduce the need for upgrade by narrowing the coupling between components. If we take the simple case of a user with a handheld computer wanting to print a document on a public printer, we can support online installation of a print driver [8]. This upgrades the handheld computer at the cost of bandwidth, time, and security concerns. Alternatively, we can query the printer in a task-agnostic, lower-layer communications mechanism, like HTTP GET, determine the set of standard types accepted by the printer, and send data to the printer in that format[11,27]. The only bit that looks like upgrade in the latter case is the query result: it takes little bandwidth, time, and it is unlikely to compromise security on the handheld computer.

Acknowledgments

Mary Baker's suggestions helped me communicate these ideas more clearly. Jean Tourrilhes directly contributed to ideas related to the ripple event discussion. I appreciate the helpful comments from Armando Fox, Gerd Kortuem, Srinivasan Seshean, Gregory Abowd, and Tim Kindberg.

References

[1] Mark Weiser, "The Computer for the Twenty-First Century," Scientific American, pp. 94-10, September 1991
[2] George Coulouris, Jean Dollimore, Tim Kindberg: Distributed Systems - Concepts and Design, 3rd edition. Addison-Wesley, 2001
[3] Nigel Davies, Hans-Werner Gellersen "Beyond Prototypes: Challenges in Deploying Ubiquitous Systems" IEEE Pervasive Computing, January-March 2002 (Vol. 1, No. 1) pp. 26-35
[4] Sameer Ajmani, "A Review of Software Upgrade Techniques for Distributed Systems" http://www.pmg.lcs.mit.edu/~ajmani/papers/review.pdf
[5] Jakob Nielsen: "Designing Web Usability: The Practice of Simplicity". New Riders Publishing, Indianapolis, 2000. ISBN 1-56205-810-X.
[6] "Thinstall packages an entire application suite into a single EXE". http://thinstall.com/ (Feb. 2004).
[7] Tim Kindberg and Armando Fox: "System Software for Ubiquitous Computing," IEEE Pervasive Computing, vol. 1, no. 1, Jan-Mar 2002, pp. 70-81.
[8] Ken Arnold (ed.): The Jini(TM) Specifications, (2nd Edition) Addison-Wesley Pub Co ISBN: 0201726173
[9] Hao-hua Chu, Henry Song, Candy Wong, Shoji Kurakake, and Masaji Katagiri, "Roam, A Seamless Application Framework", Journal of Systems and Software, Volumn 69(3), pages 209-226, January, 2004.
[10] Gerd Kortuem, Jay Schneider, Dustin Preuitt, Thaddeus G. Cowan Thompson, Stephen Fickas, Zary Segall. "When Peer-to-Peer comes Face-to-Face: Collaborative Peer-to-Peer Computing in Mobile Ad-hoc Networks." 2001 International Conference on Peer-to-Peer Computing (P2P2001), Aug 27-29, 2001, Linköpings, Sweden.
[11] Tim Kindberg and John Barton: "A Web-Based Nomadic Computing System. In Computer Networks, Elsevier, vol 35, no. 4, March 2001, pp. 443-456
[12] Dejan Milojicic, Philippe Bernadat, Rick Corben, Ira Greenberg, Rajnish Kumar, Alan Messer, Dan Muntz, Eamonn O'Brien-Strain, Vahe Poladian, Jim Rowson "Appliance Aggregation Architecture (A3)" HPL-2003-140 July 3rd , 2003.
[13] J.H. Saltzer, D.P. Reed and D.D. Clark "End-to-End Arguments in System Design" ACM Transactions in Computer Systems 2, 4, November, 1984, pages 277-288.
[14] M. Agrawal, and S. Seshan: "Development Tools for Distributed Applications" HOTOS IX, Lihue, HI, May 2003
[15] Neil Gershenfeld: "Everything, the universe, and life". IBM Systems Journal, Volume 39, 2000, p 932
[16] Umar Saif, David Greaves: "Communication Primitives for Ubiquitous Systems or RPC Considered Harmful", Proceedings of ICDCS International Workshop on Smart Appliances and Wearable Computing, 2001.

[17] Statsteam.com: “Online Games Patching Strategies” http://www.statsteam.com/Analysis.asp?ID=1, As of Oct. 2003
[18] "Windows Update is the online extension of Windows that helps you to keep your computer up-to-date" http://windowsupdate.microsoft.com/
[19] A. Russell Jones: “Is DHTML Dead? Cross-browser incompatibilities, insufficient power, and the march of technology have relegated DHTML to a small—and shrinking—niche.” http://www.devx.com/devx/editorial/16377
[20] Fox, A., Gribble, S. D., Chawathe, Y. & Brewer, E. A. , Adapting to Network and Client Variation Using Active Proxies: Lessons and Perspectives, in IEEE Personal Communications, Special Issue on Adaptation, August 1998
[21] Jean Tourrilhes. On-Demand BlueTooth : Experience integrating BlueTooth in Connection Diversity. HPL-2003-51 http://www.hpl.hp.com/techreports/2003/HPL-2003-51.html
[22] For example see references in David Lundberg: “Ad hoc Protocol Evaluation and Experiences of Real World Ad Hoc Networking” Thesis for Uppsala University http://www.update.uu.se/~davidl/msthesis/html/thesis.html
[23] Brilliant Digital Entertainment's Annual Report (Form 10KSB), Filed with SEC April 1, 2002
[24] Roy Want, Trevor Pering, Gunner Danneels, Muthu Kumar, Murali Sundar, John Light: “The Personal Server: Changing the Way We Think about Ubiquitous Computing”. Ubicomp 2002: 194-209
[25] Paramvir (Victor) Bahl, Anand Balachandran, Allen Miu, Geoffrey Voelker, Wilf Russell, Yi-Min Wang: “PAWNs: Satisfying the Need for Ubiquitous Connectivity and Location Services” IEEE Personal Communications Magazine (PCS), Vol. 6, October 2001.
[26] A. Carzaniga, A. Fuggetta, R.S. Hall, A. van der Hoek, D. Heimbigner, and A.L. Wolf "A Characterization Framework for Software Deployment Technologies". Technical Report CU-CS-857-98, Department of Computer Science, University of Colorado, April, 1998
[27] Mark W. Newman, Jana Z. Sedivy, Christine M. Neuwirth, W. Keith Edwards, Jason I. Hong, Shahram Izadi, Karen Marcelo, Trevor F Smith:. Designing for Serendipity: Supporting End-User Configuration of Ubiquitous Computing Environment. In Proceedings of DIS '02. (June 2002)