CEPIS UPGRADE is the European Journal for the Informatics Professional, published bi-monthly at <http://cepis.org/upgrade>

Publisher
CEPIS UPGRADE is published by CEPIS (Council of European Professional Informatics Societies, <http://www.cepis.org/>), in cooperation with the Spanish CEPIS society ATI (*Asociación de Técnicos de Informática,* <http://www.ati.es/>) and its journal Novática

CEPIS UPGRADE monographs are published jointly with Novática, that published them in Spanish (full version printed; summary, abstracts and some articles online

CEPIS UPGRADE was created in October 2000 by CEPIS and was first published by Novática and INFORMATIK/INFORMATIQUE, bimonthly journal of SVI/FSI (Swiss Federation of Professional Informatics Societies, <http://www.svifsi.ch/>)

CEPIS UPGRADE is the anchor point for UPENET (UPGRADE European NETwork), the network of CEPIS member societies' publications, that currently includes the following ones:
- inforewiew, magazine from the Serbian CEPIS society JISA
- Informatica, journal from the Slovenian CEPIS society SDI
- Informatik-Spektrum, journal published by Springer Verlag on behalf of the CEPIS societies GI, Germany, and SI, Switzerland
- ITNOW, magazine published by Oxford University Press on behalf of the British CEPIS society BCS
- Mondo Digitale, digital journal from the Italian CEPIS society AICA
- Novática, journal from the Spanish CEPIS society ATI
- OCG Journal, journal from the Austrian CEPIS society OCG
- Pliroforiki, journal from the Cyprus CEPIS society CCS
- Tölvumál, journal from the Icelandic CEPIS society ISIP

Editorial correspondence: Llorenç Pagés-Casas <pages@ati.es>
Advertising correspondence: <info@cepis.org>

Subscriptions
If you wish to subscribe to CEPIS UPGRADE please send an email to info@cepis.org with 'Subscribe to UPGRADE' as the subject of the email or follow the link 'Subscribe to UPGRADE' at <http://www.cepis.org/upgrade>

ISSN 1684-5285

Monograph of next issue (April 2011)
"Software Engineering for e-Learning Projects"
(The full schedule of **CEPIS UP**GRADE is available at our website)

CEPIS UPGRADE
The European Journal for the Informatics Professional
http://cepis.org/upgrade
Vol. XII, issue No. 1, February 2011

Monograph
Internet of Things
(published jointly with Novática*)
Guest Editors: *Germán Montoro-Manrique, Pablo Haya-Coll, and Dirk Schnelle-Walka*

UPENET (UPGRADE European NETwork)

CEPIS NEWS

* This monograph will be also published in Spanish (full version printed; summary, abstracts, and some articles online) by **Novática**, journal of the Spanish CEPIS society ATI (*Asociación de Técnicos de Informática*) at <http://www.ati.es/novatica/>.

"Creepy Πon i.e.", System Support for Ambient Intelligence (AmI)

Francisco J. Ballesteros-Cámara, Gorka Guardiola-Múzquiz, and Enrique Soriano-Salvador

AmI, Ambient Intelligence, environments are a challenge for building applications. These environments are data-rich, highly heterogeneous environments requiring fast systems on small devices and conventional machines, working together to build complex applications. Moreover, these applications need to be accessible from a wide range of I/O devices present in the environment, from high latency mobile networks. To build systems for AmI that work well in practice we need support them on three fronts: (i) a kernel which leverages modern hardware capabilities to move data fast, otherwise AmI environments will be slow and, therefore, dumb; (ii) a common language to exchange data and access devices, tolerating high latency links, available everywhere, otherwise intelligence will be confined to isolated components; and (iii) UIMS, User Interface Management System, capable of providing distribution, replication, persistence, and interaction heterogeneity by default, otherwise the user will need to make use of a PC to interact with the ambient intelligence. We are currently writing a new kernel, Πon ("PI-on"), leveraging the features of modern machines to provide fast services fast. Pon is designed to simplify building synthetic file systems as interfaces for AmI services, perhaps running on tiny devices. The common language of Πon is a new file system protocol, "Creepy", designed to work well despite high latency links, for both data and device access. User Interfaces, UI, for final users are to be provided by the UIMS we call "i.e.", which builds on the infrastructure provided by Πon and uses the Creepy protocol to distribute UI elements among highly heterogeneous I/O devices. This paper describes the requirements and design for these systems and, briefly, the status of their implementations.

Keywords: AmI, System Support, Creepy Πon i.e., File System Protocol, Kernel, Shared Address Space, UIMS, Zero-copy.

1 Introduction

An AmI, Ambient Intelligence, environment needs to sense the user needs and respond to them in real time. In

Authors

Francisco J. Ballesteros-Cámara received his MS in Computer Science on 1993 and his PhD in the same matter on 1998, at the *Universidad Politécnica de Madrid*, Spain. While an undergraduate, he received several grants from European research projects where he developed run-time software for programming languages. Later, he worked for a number of years in telecommunications companies producing systems software (he is the co-author of LiS, a STREAMS framework for Linux.) Since 1995 he has been a Lecturer at several Spanish universities where he has been teaching and developing Operating Systems. He developed the Off++ kernel, for the 2K Operating System jointly with the SRG at University of Illinois at Urbana Champaign. 2K evolved later into the Gaia OS. Since then, he has been developing Plan 9 from Bell Labs software. Recently, he developed Plan B and Octopus, two different ubiquitous systems derived from Plan 9 from Bell Labs that he uses to perform daily computing tasks. He is the head of the Systems Lab, <http://lsub.org>, where Pon is being developed. Additional info can be found at <http://lsub.org/who/nemo>. <nemo@lsub.org>

Gorka Guardiola-Múzquiz obtained his MSc in Telecommunication Engineering from the *Universidad Carlos III de Madrid*, Spain, in 2003, and his PhD from the *Universidad Rey Juan Carlos*, Spain, in 2007. He is currently working as a full time lecturer at the *Universidad Rey Juan Carlos de Madrid*. He has been an intern at Bell Labs NJ, Bell Labs Antwerp and Austin Research Lab (IBM), where he has worked on Plan 9 related projects, including security for the femto base station, an implementation of IKE and layered block devices. His research areas are operating systems and ubiquitous systems. He was part of the team developing Plan B and Octopus operating systems. He has been working in R&D on both the Plan 9 from Bell Labs, Inferno and the Octopus Operating Systems. He is a member of the Operating Systems Lab, <http://lsub.org>, where Pon is currently being developed. <paurea@lsub.org >

Enrique Soriano-Salvador obtained his MSc in Computer Science from the *Universidad Rey Juan Carlos de Madrid*, Spain, in 2002. He had been awarded a four-year F.P.I. full time grant from the Spanish Ministry of Science and Technology. and his PhD from the *Universidad Rey Juan Carlos* in 2006. He is currently working as a full time lecturer at the same university. As a postgraduate, he worked as software developer for the said university and as a UNIX trainer before becoming a member of the Systems Lab. His research interests include security, operating systems and pervasive computing. He was part of the team developing Plan B and Octopus operating systems. He has been working in R&D on both the Plan 9 from Bell Labs, Inferno and the Octopus Operating Systems. He is a member of the Operating Systems Lab, <http://lsub.org>, where Pon is currently being developed. <esoriano@lsub.org>

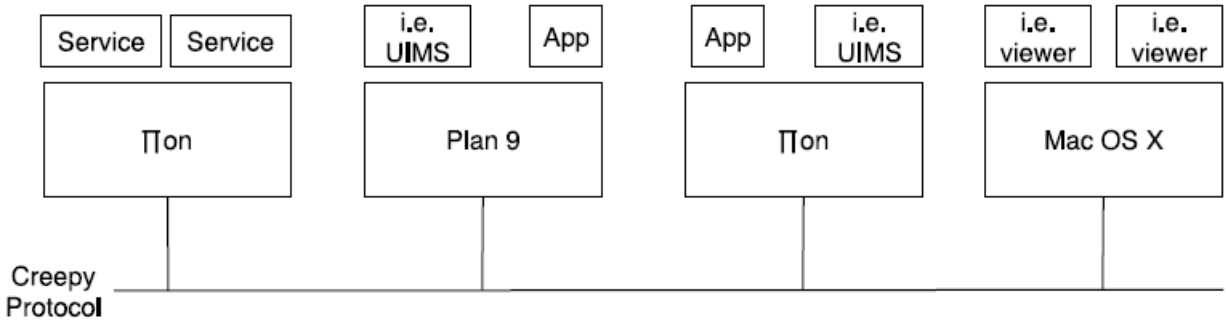

Figure 1: The Creepy Pon Environment: Heterogeneous Systems run Different Services for AmI, Pervasive Applications, and i.e.. UIMS and Viewers.

order to do so, it needs to process data from sensors, some of which are bandwidth hungry, such as video input. Then, it needs to integrate them with models and heuristics for the user behaviour. Finally, it needs to respond by controlling actuators and user interfaces. All this must be done within a real time feedback loop while interacting with the user. Latency is a critical parameter for the whole system to work.

As a result we need servers, protocols, and user interfaces customized to work well in this data-rich environment. Most work done in the past to this effect has been fragmentary, with special customized hardware, applications, and services, but not thinking of the system as whole. More than just focusing on some special services or applications, we need to focus on how to integrate everything into a whole system that the user can perceive as the "Environment".

In the past, we have built different systems and protocols for pervasive computing, including Plan B [1][2][3], Octopus [4], Omero [5], etc. We are now using what we learned from building them to provide appropriate systems services to bring intelligence to the environment.

In particular, we are focusing on three specific components that we consider critical in this respect:

1. A new system kernel for machines providing AmI services.
2. A new file system protocol to let them speak to each other.
3. A new UIMS technology to let users interact with the environment.

Current general purpose kernels are not optimized for serving data fast, especially since recent advances in hardware have rendered many of their assumptions obsolete. **Πon**, or "PI-on", our new kernel, considers that the common case is either a tiny low-end machine (e.g. a small ARM machine for embedded sensors) or a powerful multi-core computer with copious resources.

The abstractions it provides take both cases into account, aiming at being able to pipe data fast, with low latency, from/to devices embedded in the environment.

Pon uses synthetic file systems as its main abstraction, as our previous systems did [3][4]. Synthetic file systems have proved to be a very general abstraction on which to build distributed systems [6][7]. They address many of the problems of distribution, including protection, naming, access transparency, heterogeneity, and concurrency control.

However, existing protocols for distributing file systems can be slow, especially in the presence of high latency networks like mobile networks (where latency is due to retransmission and sophisticated channel coding). Moreover, most of them are not suitable for accessing devices represented by synthetic files (e.g. devices under /dev in Linux).

Creepy is a new file system protocol specially designed to export synthetic files through the network on high latency environments.

Previously, we developed several file-based UIMS (Omero [5] and O/Live [8]) that transparently permit the distribution and replication of UI on heterogeneous I/O devices. We can take advantage of the new kernel and protocol to improve upon them, and the result is a new UIMS called **i.e.**, based on synthetic files and persistent event channels.

i.e. is well adapted to AmI, because it decouples the application from its user interface, both from the bandwidth point of view and from the adaptation point of view. Interactions are confined to the user interface whenever possible, and only very high abstraction messages cross the network.

The resulting environment is depicted in Figure 1. In what follows, we describe the requirements, design and, briefly, the implementation status of each of these components.

2 Πon

Most of the assumptions behind current general purpose OS architecture have changed:

- The virtual space now is much larger than it used to be. Even though pages or more specifically TLB entries may be a scarce resource, the address space itself is not.
- Concurrent programming is a must, in and out of the kernel. Cores are not getting faster, in fact, sometimes the opposite; they are just growing in number per machine. Concurrency requires automatic garbage collection, otherwise making correct programs is difficult.
- Because of the way the multicore architecture is built,

> “An AmI, Ambient Intelligence, environment needs to sense the user needs and respond to them in real time”

it is very easy for one process to poison the cache of another. Particularly the L1/L2 which are usually local to the processor. Spin locks and, in general, memory locking operations where processors alternate locking the access to a memory area, are a bad idea in this environment without special provisions. Playing ping-pong with the L1/L2 cache is specially bad for performance. Processors tend to grind to a halt on memory coherency protocols and the software runs slow as a consequence. Which cache limits us most, the data or the text, is still something we need to measure carefully.

- In most machines, RAM memory is also huge, spanning tens of GB, but caches have not grown as fast as RAM and are now scarce resources.
- On the other hand, there are small machines (e.g. ARM machines) designed for embedded systems. They are not different from a single core in desktop computers.

Considering these points, it is desirable for a kernel to include the features described next. We are building a kernel, Пon, which does so.

2.1 Пon Design

Considering that most I/O devices are capable of performing gather/scatter, a zero-copy framework can move data by copying pointers and not data itself.

In order to exploit concurrency on multicore machines, applications need garbage collection. A buffer framework supporting zero-copy is an opportunity to provide it, relieving the user from having to manually manage memory or having runtimes to do so. Pon is designed to include an immutable buffering scheme similar to the one in the io-lite [9], but intended for fast communication in modern multicore machines and providing automatic garbage collection. Following [10] we plan to make our protection coarser than it currently is, sharing a big part of the address space between processes. This will enable us to use this channels to pass references around.

Because machines are ccNUMA these days (each core has its own cache attached and communicates with different cores by using a hypertransport bus), it is important to rely on message passing as the primary communication and synchronization mechanism. Pon includes message passing and channels to communicate data between processes and also between processes and the kernel. ** These channels are typed, and protection for the referenced data changes when a message is sent through them. **

Considering the previous point, scheduling should be closely tied to communication. For example, the processor might be handed directly from sender to receiver without going through the kernel. The aim is to keep the data cache warm. At the same time, the concurrency model can build on this.

For latency, channels can be buffered in order to have the receiver (kernel or process) asynchronously running in another core, batching operations to reduce the number of context switches in a similar manner to FlexSC [11].

When scheduling, there is always a compromise between CPU usage and cache usage. If a process is tied to a processor, it is likely that the entries in the caches can be reused by using some form of tagging with process identifiers. However, if all the processes are tied to a subset of available processors, we may be wasting resources. The tradeoff becomes harder if we consider other effects not described here.

Пon uses a Warp Scheduling Drive inspired by Apertos computational field [12]. A curved space of costs can be set for scheduling, where highly coupled processes find themselves inside a potential well, and are unable to move to another processor; whereas non interactive processes are free to move, although they may find a potential barrier to using processors depending on the cache state. The idea is that the system automatically computes the forces in the field by self measuring, and moves processes accordingly. Not only caches but other shared resources in the processor can be taken in account as well. For example, hyperthread (HT) cores share more resources with each other than real cores. Processes running in the same HT core can repel each other so that one of them migrates to a proper core.

In modern computing, shared libraries [13] have a large cost in execution time because of the overload added by dynamic linking, and the size of the linker and symbols themselves. More importantly, they can have a great cost in manageability [14][15]. However, if many instances of the same library are running on different processes (with different binaries), RAM and, more importantly, cache misses, can be saved. Pon gets the same benefits by using a new approach: shared, but statically linked, libraries (SSLL).

In SSLL, we have static linking, but the library is shared between the different binaries. Each binary is statically linked against an immutable version of the libraries required. Each library is assigned a global, unique portion of the virtual address space. Therefore, different libraries occupy different portions of the global virtual space, no matter the

> “**Creepy** is a new file system protocol specially designed to export synthetic files through the network on high latency environments”

> “In modern computing, shared libraries have a large cost in execution time because of the overload added by dynamic linking”

binaries and libraries are to be paged on demand. The abundance of virtual addresses is used to distinguish between libraries.

3 "Creepy", a New Distributed File System Protocol

The new distributed file system protocol called *Creepy* is intended to become the Πon file protocol. Creepy supports a distributed file system for the Internet, capable of working well to access devices both on the local and wide area; therefore with caching in mind. This is desirable for bringing together devices and services as required for building AmI environments.

We need a file system protocol supporting access to distributed files and devices. For this, 9P [16] or Styx [17] suffice. However, we have extra requirements that make 9P not so well suited for our purposes. This is the full list of requirements:

- The protocol must support access to synthesized files and devices, as we want it to reach sensors, actuators, and software devices in general. 9P and Styx meet this requirement, but not the following ones.
- The protocol must work well across high-latency and wide-area, network links. Otherwise we would not be able to use it on mobile networks, from overseas or from our (poorly connected) homes.
- The protocol must permit client disconnections, without discarding the client state. Otherwise, suspending clients would be impractical or require additional software.

3.1 "Creepy" Design

To work well with devices, the protocol must support file descriptors as found on clients. Just having request and responses send names or resolving them on demand is not enough. There has to be a state in the server per open file to be able to keep the state of the client, e.g. the offset. This state is indexed by a file descriptor. The equivalent concept in the protocol is the *fid*, as used in 9P [16]. Like in 9P, a *fid* is a small number used as a file identifier. Unlike in 9P, the server assigns *fid* numbers on behalf of the client. Clients must maintain their own data structures to keep track of *fids*, and they do not usually look them up by *fid* number, which means that it is better to let the server assign them; the server can use the *fid* number as an index and save a hash table.

Support for disconnected operation requires the server to be able to keep track of *fids* in use by a client that is disconnected. This is achieved by using "sessions", a container for the fids of the client. A session is established prior to the Creepy dialogue for file access. In some cases creating a new session; in other cases associating to a session from a previous connection. Sessions may be collected by the server (and their state released) after a disconnection, after some time past the disconnection, or after a server reboot. When to do it is suggested by the client. The server decides.

To reduce latency effects, and utilize effectively each round trip, protocol requests exchanged between clients and servers are batched. Each batch behaves as an RPC but, instead of using a single request and a single reply, a batch consists on a series of individual operations, each one with its request (and response).

To clarify, a batch, or RPC, is a series of operations, but does not need to be a single message (i.e., a single "write" on the channel). Each operation is a separate request but it is part of a logical RPC, or batch.

A client sends a series of requests (one per operation) for a single RPC, and then waits for a series of response messages from the server. For each operation there is a request message (or "transaction" message, or T-message) and a corresponding reply message (or R-message). This is similar to 9P.

Requests to agree on a protocol version, to agree on a session to use, and to flush outstanding requests, are RPCs on their own. But all other requests must be made within a batch of requests.

A batch (or RPC) has a *begin* request, followed by one or more requests, followed by a final *end* request. This has important implications. A server is free to read from the network all requests in a batch, without starting any of them before reading the end of the batch. Because all such requests are considered a single RPC, with several operations in it, an error in one operation means that the rest of the batch is ignored by the server; the final error response effectively terminates the RPC when seen by the client.

Batching is needed to achieve a single round trip time in those cases where the client, or an intermediate cache, wants to perform a series of operations and does not need the result of any of them before issuing the next one.

A batch is not packaged as a single message, in order to permit multiplexing of the communications channel. Should a batch be a single message, no requests from other clients

> “For enabling AmI environments, it is paramount to provide applications with multimodal, replicated, distributed user interfaces (UI)”

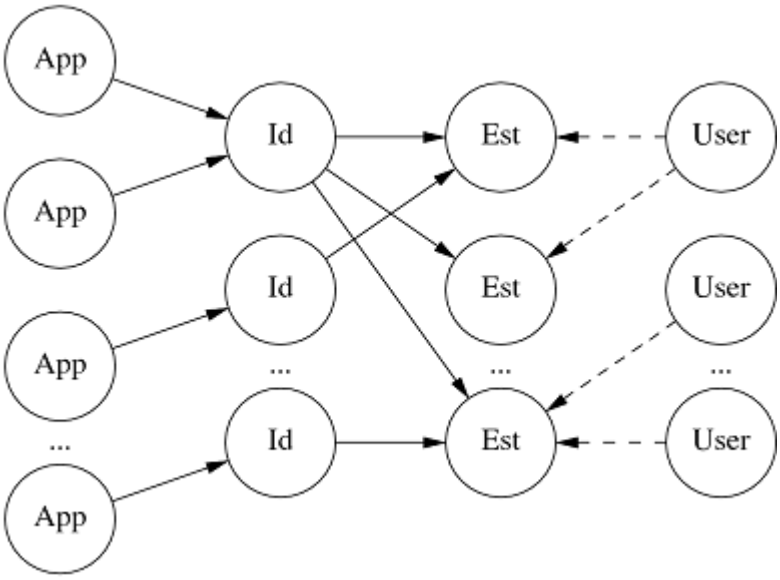

Figure 2: Applications interact with their *Id.* There is a *n* to *m* mapping between *Ids* and *Ests.* User interaction is confined to *Est.* Application interaction is confined to *Id.*

may be interleaved with it, and a long batch would effectively block other clients sharing the channel. Packaging each different operation as a different message is also beneficial for the implementation, which can pack, unpack, and handle all messages in the same way. To help intermediate caches to decide what to batch when they need to ask the server for data, each batch speaks about a single *fid*. An intermediate cache may read a series of requests (from a single batch) and, when all of them are known, decide if it makes sense to forward all of them at once to the server. It might decide instead to serve a prefix of the batch from local data, forward the rest to the server, and then reply to the client. Other approaches, mixing requests in a more general way, make it not clear for the cache how long to wait before asking the server: asking too soon leads to multiple round-trips, asking too late leads to extra delays in the service.

A client (or an intermediate cache) must know something about the semantics of each file, to know how to cache it, if at all. There are several types of file to consider:

- Conventional files. No extra requirements on them.
- Device files. They should not be cached at all. All requests on them are to be forwarded to the final server, or the semantics will change and the device interface will break.
- Append only files. These are only appended, which may be exploited by clients and caches regarding what to cache.
- Timely files. These correspond to audio and video media, where it is usually more important to deliver data timely than it is to retransmit every piece of data.
- Read-only files. Immutable files have nice properties for caches and clients. Those files that are meant to be read only for their whole life, belong to this category. For example, system binaries. Previous versions of mutable files may be considered also immutable.
- Synchronous files. For some files, it is important for clients to see all writes made before (i.e., UNIX semantics). In all other cases, it may suffice to see all writes made prior to the last close of the file (i.e., session semantics). The later is better for caching and works fine in most cases; thus, it is a desirable default. However, the former is sometimes necessary and, therefore, some files may have to be flagged as "synchronous".

Being able to recognize different types of file, as enumerated, is essential for AmI environments, where some files may correspond to device interfaces, others might represent logs of events (perhaps append-only files), yet others might be conventional files.

4 "i.e.", a New User Interface Management System

Having a fast kernel designed for providing AmI services, and a Creepy protocol to bring components together, does not suffice.

For enabling AmI environments, it is paramount to provide applications with multimodal, replicated, distributed user interfaces (UI). Regarding the UIMS and the window system, this implies the following:

- Applications must be able to create interfaces independent of the technology used for I/O. In many cases, graphics and voice may be equivalent (e.g., a menu can be shown using graphics or may be presented by reading options). The application should remain unaware of these details.

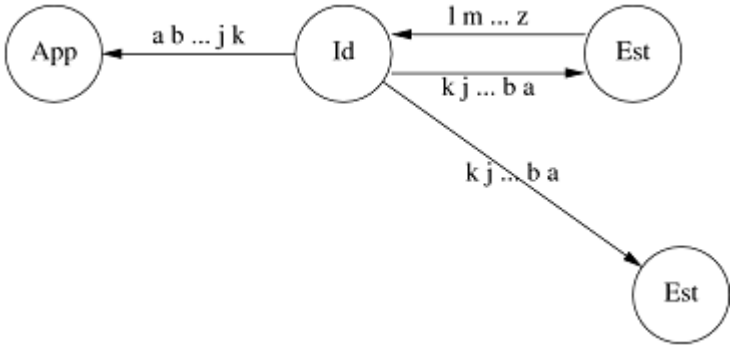

Figure 3: *Est* may perform speculative operations, streamed to *Id.* They are considered real when echoed back from *Id.* Here, operations from *a* to *k* are considered stable. Operations *l* to *z* are considered as mere speculation, and *Est* must be prepared to undo their effect if something different from *l* is seen after the echo for *k.*

■ Applications should not be concerned with how many copies of their UIs are deployed on the environment. It should be feasible to make a duplicate of (part of) a UI to access it from more than one place at once.

■ Interfaces should not be *blocks*. They are made of parts (*widgets*, or UI elements). There is no reason why UI elements cannot be handled independently. They might be rearranged by the user, or perhaps replicated (without replicating the rest of the interface.

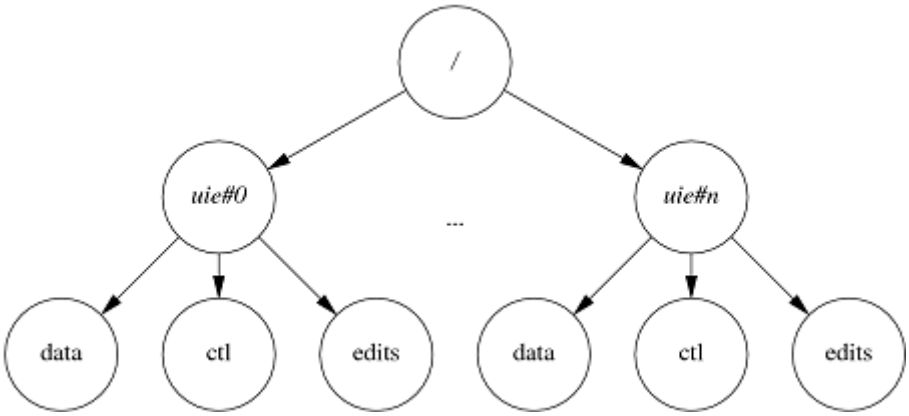

Figure 4: *Id* is a file server that provides a file interface for User Interface Elements (IE). Each IE appears to be a directory containing several files used to operate on it.

■ How interfaces are rendered, or presented to the user, should be independent of the structure of the UI as created by the application. Depending on the device, a set of interfaces might be presented one-at-a-time, or perhaps using tabs, or windows, or by a 3D representation, or by any other means a viewer (or presenter) for the interface deems reasonable.

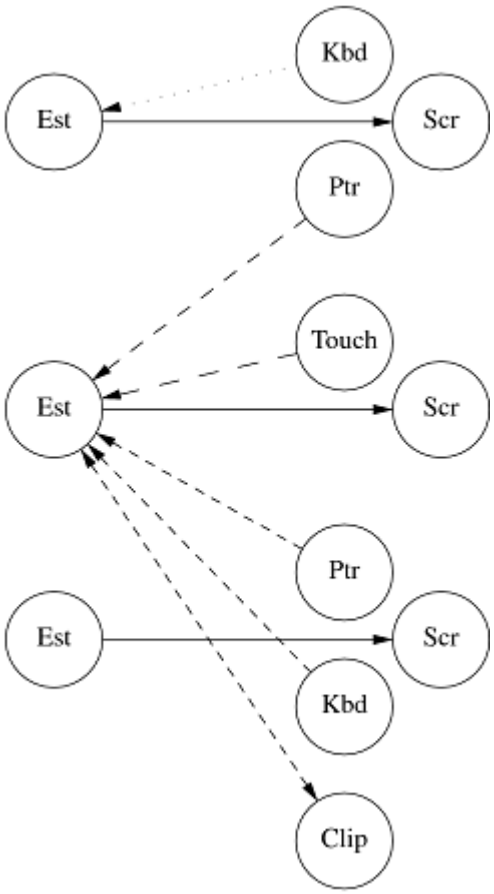

Figure 5: I/O Devices. There is an *Est* instance per output device. Input devices form groups (represented by arrow types in the figure) and may be directed to any *Est* instance. Clipboards operate in the same way.

■ I/O interactions should be confined to the viewer or presenter at hand. Otherwise, editing or interacting with the interface would require a permanent connection to the application or to the UIMS, which is not always reasonable.

■ Operations on an interface should not require much regarding throughput and latency of the communication link used to reach the UIMS and the application. Inserting a line on a text could require sending a few bytes, with the new text, but it is not reasonable to require all parties involved to reside in a well-connected network to be able to achieve this.

■ UI elements should be able to move around, as requested by the user, perhaps crossing machine boundaries. Otherwise we wouldn't be able to move UI elements to other displays or I/O devices.

■ The state for the session, as seen by the user from one location (and perhaps after moving to another and recalling it), should persist. There is no need for users to recreate their preferred session once and again each time they "connect" to the system.

■ Concurrent interaction with multiple copies of an UI element must be feasible, perhaps overseas. Otherwise, replication of UI elements would not be actually perceived as a replication. If a button or text is shown in three different places, it must be feasible to edit it from all three places, at the same time. It is likely that some of them would be co-located, and that the user chooses one of them at one time, and a different one next, at will.

These requirements are feasible today, given the technology available. Current window systems, including research ones, fail in one or more points. The two previous UI services we implemented for pervasive computing, Omero [5] and O/Live [8], achieve some of them, but not all. A descendant of these systems, called **i.e.**, is being built in the hope to meet all of them.

“The work described here builds on the previous operating systems we have built for ubiquitous computing and AmI: Plan B and Octopus”

4.1 "i.e." Design

i.e. is both a UIMS and a window system. It is designed by dividing the tasks in two different parts:

- **Id:** This part is the UIMS. It is responsible for keeping the state of the UIs created. At one time, there will be many copies of *Id.* Perhaps running at different machines.
- **Est:** This part is the UI viewer. It is responsible for taking different UIs (or parts of them) from one or more *Id* instances, and presenting them. All user interaction is confined to *Est.*

It should be understood that at one point in time, multiple copies of *Id* and multiples copies of *Est* would be running in different machines.

In essence, the task performed by *Id* is to receive an abstract stream of updates to UI Elements, and to multicast it to *Est* instances interested on the elements (see Figure 2). At any point in time, *Est* may have performed several updates to UI Elements shown. Some of them have been received by *Id* and echoed back to *Est*, some of them (following the formers) not. The latter are still speculative changes, and, due to concurrent updates performed by several *Est* or by the application, might be rejected if they are not echoed back in the order expected (for example, if an external insertion on text is seen by *Est* when it was expecting the echo for one of its own edit operations).

In Figure 3, both *Est* components are presenting a particular element. The one above has been updating it (due to user interaction) and has performed updates from *a* to *z.* Only updates from *a* to *k* have been echoed back from *Id* (from the one supporting the element considered). Those are seen by everyone, including the application. Updates from *l* to *z* are still speculative and presented only from the *Id* above. As shown in the figure, the application is not different from *Est* regarding delivering of updates.

4.2 Interfaces

Id is a synthetic file server. Its main interface to other programs, including *Est,* applications, and external programs, is a synthesized file tree. UI elements are represented by synthetic files. This is the same approach followed by the predecessors Omero, and O/live [5][8].

Programs operate on interface elements kept by *Id* by operating on the files it implements. Figure 4 shows an example file tree served by *Id.* It is important to notice that these are not actual files. Instead, they are just the interface for the abstractions implemented by *Id.*

If Interface Elements (IE, in what follows) files are accessed by using a network file system protocol designed to avoid latency problems, like `Creepy, i.e.` can perform well even through high latency WANs, as its predecessor O/live [8] does.

4.3 Input Devices and Clipboards

Input devices and clipboards are independent of *Id* and *Est*, but cannot be left apart.

As shown in Figure 5, there may be different input devices involved. The same happens to output devices, but deploying one *Est* instance per output device suffices to coordinate them through the *Id* instances involved.

Input devices are connected to *Est*, because they are usually close to the terminal used for I/O. Unlike on other systems, we consider I/O devices as stand-alone devices. Each device is able to send its output as input to one *Est* or to another. At any point in time, the device may change its connection and address a different *Est.*

For the sake of convenience, I/O devices may be grouped so that redirecting one to one *Est* also redirects the others in the group, in the same way. Groups are named so that different *Est* implementation might choose, for example, to show different pointers for different pointing devices attached, but only one per I/O group.

At a given point in time, there may be multiple input devices attached to a given *Est.* Suppose that in the scenario depicted in figure 5, the machine running the middle *Est* seems to be suffering some activity. The pointer device from the top-most machine is directed towards the middle *Est.* However, the keyboard device close to it is left alone for use on the top-most *Est.* Devices from the bottom-most machine are all directed towards the middle *Est.*

Clipboards are handled in the same way, like any other input device. They differ just in that they are bi-directional. They can send input to *Est* as well as they can accept output from it. For example, cutting some text with a mouse or multitouch associated to a given editing device would copy the text into all clipboards associated to such edit device.

5 Related Work

The work described here builds on the previous operating systems we have built for ubiquitous computing and AmI: Plan B [3] and Octopus [4].

Plan B [3] is a peer to peer operating system which can adapt to changes on the environment by mounting dynami-

“A descendant of two previous UI services, Omero and O/Live, called **i.e.**, is being built in the hope to meet all of the requirements”

cally resources as they appear on the network. Plan B is a descendant of Plan 9 [6] and, in the same manner, uses synthetic files to model resources. The Plan B UI, Omero [5], is an ancestor of i.e., and therefore related. However, Plan B used a kernel built using abstractions that were designed for hardware as it was in the 80s (as most other popular OSes do). Pon uses ideas from different existing kernels, like immutable buffers [9], system call batching and the Apertos computation field [12] but adapting them to present hardware.

Octopus [4] is a descendant of [7] and can run hosted on different operating systems. The Octopus protocol, Op [18], an evolution of 9P [16] and Styx [17], designed to work on high latency connections. The problem is that Op loses the connection whenever the network has a transient failure. Creepy is a descendant of Op, designed to fix this and other problems. O/Live [8], the UIMS for Octopus is also an ancestor of i.e. and O/Live inherits from Octopus the problem with transient network failures, and therefore we had to design i.e. to address this and other problems.

6 Conclusions

In this paper we have identified three different critical components for building AmI environments (specialized kernel support, file system protocols, and UI support). We have enumerated different requirements that, after our experience in the field, we have found to be relevant and, therefore, should be considered for building them. We have also described a set of design guidelines for each one, which we are following to implement prototypes for them.

As of today, we have a prototype for **i.e**. almost finished, a implementation for the **Creepy** protocol finished, a prototype for a Creepy file system (speaking the protocol) near to be functional, and have just started the work on **Πon**. In the near future we will bring these three components into operation to port our smart space (i.e., AmI) into a more powerful environment; to evaluate the result and measure the benefits or drawbacks of the ideas presented here when used in practice.

References

[1] F. J. Ballesteros, K. Leal, E. Soriano, and G. Guardiola. "Plan B: an operating system for ubiquitous computing environments," in IEEE PerCom 2006., 2006.

[2] Ballesteros, F.J., Soriano, E., Guardiola, G., and K. Leal. "Plan B: Using Files instead of Middleware," IEEE Pervasive Computing, pp 58–65, 2007.

[3] F. J. Ballesteros, K. Leal, E. Soriano, and G. Guardiola. "The Plan B OS for ubiquitous computing. voice, security, and terminals as case studies." Elsevier Pervasive and Mobile Computing Journal, vol. 2, pp 472–488, 2006.

[4] F. J. Ballesteros, P. de las Heras, E. Soriano, and S. Lalis. "The Octopus: towards building distributed smart spaces by centralizing everything," in UCAMI 2007, 2007.

[5] F. J. Ballesteros, G. Guardiola, K. Leal, and E. Soriano. "Omero: Ubiquitous user interfaces in the Plan B operating system." In IEEE PerCom 2006., 2006.

[6] R. Pike, D. Presotto, K. Thompson, and H. Trickey. "Plan 9 from Bell Labs," in In Proceedings of the Summer 1990 UKUUG Conference, 1990, pp 1–9.

[7] S. Dorward, R. Pike, D. Presotto, D. Ritchie, H. Trickey, and P. Winterbottom. "The Inferno operating system," Bell Labs Technical Journal, vol. 2, no. 1, pp 5–18, 1997.

[8] F. Ballesteros, E. Soriano, and G. Guardiola. "Towards persistent distributed, and replicated user interfaces in the Octopus," in Intl. Workshop on Plan 9, Murray Hill, Bell Labs, NJ, 2007.

[9] V. Pai, P. Druschel, and W. Zwaenepoel. "IO-Lite: a unified I/O buffering and caching system," ACM Transactions on Computer Systems (TOCS), vol. 18, no. 1, 2000.

[10] J. Chase, H. Levy, M. Feely, and E. Lazowska. "Sharing and protection in a single address space operating system," ACM Transactions on Computer Systems (TOCS), vol. 12, no. 4, 1994.

[11] L. Soares and M. Stumm. "FlexSC: Flexible system call scheduling with exception-less system calls," in USENIX OSDI, Vancouver, Canada, 2010.

[12] Y. Yokote. "The Apertos reflective operating system: The concept and its implementation," in ACM, OOPSLA, 1992.

[13] R. Gingell, X. D. M. Lee, and M. Weeks. "Shared libraries in SunOS," in USENIX, San Diego, California, 1987.

[14] S. Eisenbach, V. Jurisic, and C.Sadler. "Feeling the way through DLL hell," in Proceedings of the First Workshop on Unanticipated Software Evolution (USE 2002), Malaga, Spain, 2002.

[15] B. R. Holt. "Do solutions have to be so complex?" IEEE Software, vol. 21, pp 8–10, 2004.

[16] AT&T Bell Laboratories. "Plan 9 programmer's manual, section 5: Plan 9 file protocol," Murray Hill, NJ, 2002.

[17] D. R. R. Pike. "The Styx architecture for distributed systems," vol. 4, 1999, pp 146–152.

[18] F. J. Ballesteros, G. Guardiola, E. Soriano, and S. Lalis. "Op: Styx batching for high latency links," in In Proceedings of the International Workshop on Plan 9 2007, 2007.

