Plan B: Boxes for networked resources

Francisco J. Ballesteros, Gorka Guardiola Muzquiz,
Katia Leal Algara, Enrique Soriano
Pedro de las Heras Quirós, Eva M. Castro,
Andres Leonardo, and Sergio Arévalo*

Laboratorio de Sistemas
Universidad Rey Juan Carlos
Madrid, Spain.
ls@lsub.org

Abstract

Nowadays computing environments are made of heterogeneous networked resources, but unlike environments used a decade ago, the current environments are highly dynamic. During a computing session, new resources are likely to appear and some are likely to go offline or to move to some other place. The operating system is supposed to hide most of the complexity of such environments and make it easy to write applications using them. However, that is not the case with our current operating systems. Plan B is a new operating system that attempts to allow the applications and their programmers select and use whatever resources are available without forcing them to deal with the problems created by their dynamic distributed and heterogeneous environments. It does so by using constraints along with a new abstraction used to replace the traditional *file* abstraction.

Keywords: Distributed systems, Operating Systems, Adaptability, Pervasive computing.

1 Introduction

The computing environment used to write this paper is made of three different network technologies (ethernet, wireless ethernet, and serial links) that interconnect a number of different devices including laptops, hand-held pocket PCs, desktop PCs and a file server. Some of these machines have large displays, some do not. The same happens with keyboards, audio devices, disks, and other resources. Furthermore, resources available to a human or an application using the system greatly vary upon time because the devices can move. Besides, other resources like printers, scanners, etc. should be used or not depending on the location of the user and their operational status (they go offline some times).

To pick up an example to illustrate the need of a new operating system, consider what a user or an application[1] has to do for printing a file when using the pocket-pc. We would like to be able to say just "copy this file to a printer" and let the system discover if there is a printer at hand willing to accept print jobs. If there are several printers available, we would like the system to choose any one that understands the format of the file to be printed. If no printer understands the file format but there is a program to convert the data to the printer format, we would like the system to run that program and then queue the result for printing. This is not the case with current operating systems. Furthermore, existing systems are likely to send the file from the file server to the pocket-pc (which can be connected through a very slow link) just to send it again to the printer. This means that once more the user has to make the job of the operating system by selecting a machine well-connected to the printer where to execute the print command.

We believe that we have enough technology to be able to do all this with just a single command like

```
cp /this/file /any/printer
```

but in the operating systems we use it turns out to be much more complex. This may be a symptom that existing operating systems are not supplying appropriate services to handle our computing resources.

*This work financed in part by Spanish MCYT TIC-2001-1586-C03-01 and URJC PPR-2003-40.

[1] In what follows we use the term "user" to refer both to users and to applications using the system.

To define the problem more clearly, we can say that existing operating systems are not helping much their users to select which resources to use. Note that this problem, which could be named the *"resource selection problem"* is different from the problems of both locating and discovering resources, and is actually a simplified version of the problem of taking context into account while considering user requests.

This problem can be seen in plenty of different examples, whenever the user is selecting a particular resource among the ones available in the network. For example, we would like to say "execute a program" without taking care of which binaries are available for the program, which architectures they can execute on, and which processors of such architectures are available. Programs like editors would like to ask the operating system to "save data to a temporary file", and let the system discover whether to use a local file system (if any), or the department's file server (if available), or any nearby disk willing to accept requests for temporary storage from our machine (when available). If we are lucky, our operating system would allow us to use resources from the network but it would still leave up to the user the task of selecting which ones to use even when the choice is obvious.[2]

A different, but related, problem is that once the resources are selected, we may change our mind. For example, many of us have wanted to be able to use our mouse for a while to help a colleague sitting in the next desk, instead of having to stand up and use his/her mouse. We would also like to use a keyboard from a desktop machine to type on a networked pocket PC with no keyboard. Although the application considered has already a mouse or a keyboard to receive events or characters, we may still want to make it use different devices for a while. Another instance of the same problem is that, due to changes in the network, an application using a network connection may be forced to switch to a different connection to stay connected (e.g. switching from a tcp stream to an infrared connection). We refer to this problem as the *"resource redirection problem"*.

The objective of this work is to provide a computing system where applications could use the plethora of networked resources without dealing with the complexity of the environment by themselves; more precisely, to build a system that addresses both the resource selection and redirection problems on behalf of the applications. The system has been built and is named "Plan B".

In what follows, section 2 shows the main ideas behind Plan B. Sections 3 and 4 give an overview of the system and its main elements. Then we show how such elements are used to address the problems faced in sections 5 to 7. Sections 8 to 10 discuss how we address some important issues like heterogeneity, failures, garbage collection, and protection. Section 11 shows some implementation details. Section 12 discusses more examples to show how the system works. Section 13 explains the lessons we learned while building and using the system. Sections 14 and 15 discuss related and future work.

2 Plan B

The main new idea in Plan B is the introduction of a new abstraction, the box [3], designed to let the application use networked resources in an easy way. But we believe that its main contribution is not any idea in particular, but how the combination of its design principles makes up an environment that is more simple to use for today and upcoming computing resources. These are the principles Plan B is built on:

- All resources (processes, devices, etc.) are perceived as a single abstraction, the *box* [3]. Boxes are typed data containers that are operated using a `copy` operation (instead of the traditional read/write interface used for files) and have *constraints* that determine how they can be used together. This lets Plan B know which resources are being requested to be used together (a binary is copied to a processor, a file to the printer, etc.) and which constraints must be considered when using them (binaries must match the processor architecture, file formats must match those understood by the printers, etc.). The box abstraction is further discussed in the next section. Note that traditional read and write operations can still be performed by copying into or out of on-demand created boxes that represent the application's memory. This is further illustrated later.

- The system operates on both local and remote boxes through the same protocol, called *Op*. Any server on the network implementing this protocol can provide boxes (i.e. services) to be used from a Plan B process.

- Name spaces bind names to boxes. Each application has its own name space and can customize it. Customization is done by defining names for

[2] It is a matter of taste, but for the author this case is when all machines involved run either Plan 9 or Inferno. Both operating systems export *all* resources to the network using a file interface, which at least is more than other systems do.

boxes, as well as the order in which they should be searched. More than one box may be bound to the same name.

- Box operations use names instead of descriptors. Therefore, applications keep no connections to resources, they use the network to send self-contained requests. This is important to better tolerate changes in the network.

- Boxes can be advertised to the network as they become available. Applications can instruct their name spaces to automatically import (i.e. bind) new boxes as soon as their advertisements are received. Applications use this mechanism to learn of resource availability and to adapt the name space to the actual environment.

These principles lead to a simple system with just 14 system calls that includes all the functionality of a typical operating system. The complete list of system calls is as follows:

System call	Purpose
cast	Defines a type conversion
change	Changes the current box name
chinfo	Modifies metadata for a box
copy	Copies one box to another
delete	Deletes a box
dot	Retrieves the current box name
forget	Forgets about an imported box
import	Defines a new name for a box
info	Returns metadata for a box
kind	Returns the box type/constraints
link	Links one box to another
make	Makes a box
selectors	Returns inner box names
uncast	Forgets about a type conversion

Plan B owes much to the design of Plan 9 [14], which also uses a single abstraction, the file, to export all resources to the network. But unlike files in Plan 9, boxes permit Plan B to take over the task of selecting and combining the resources needed by the user, at least most of the times.

3 The Box

A box [3] is an abstraction which is meant to replace the more traditional file abstraction and tries to capture enough of high-level data semantics and relationships to solve the problems faced.

A box contains data and may also contain inner boxes, leading to a tree structure. Boxes in the tree are named using path names similar to the ones used in file systems, with components separated by slashes. There is no difference between boxes that contain inner boxes and those that do not. All boxes are operated with the same set of operations. Most of the Plan B system calls are just the box operations shown in this section.

There is no `open` operation in Plan B. Box operations use box names, which means that a box name is likely to be resolved every time a box is used. Although this adds some performance penalty, this is crucial in that it allows Plan B to decide to which resource the name should be resolved on each operation.

The most important operations to handle boxes are `copy` and `link`. The first one conceptually copies data from a box to another. By supplying `copy` instead of `read` and `write`, the interface permits the system to be aware of which two resources are being combined. The second one, `link`, is used to tie two boxes together. `Link` was introduced because it is a convenient way to express that a box should be like another one. The implementation of `link` determines that the box linked is either a reference or a replica of another box. The exact implementation (reference or replica) of `link` depends on the boxes linked. References are useful to address the resource redirection problem, replicas are useful to maintain copies of resources at different locations so that the system could select among them.

The operations are atomic with respect to other operations on the same box. This means that multiple operations on the same box do not overlap.

We have seen most of the box interface; other operations are `delete`, which removes a box, `info`, which retrieves box metadata, `chinfo`, which updates box metadata, and `selectors`, which asks for the names of inner boxes to a given box. Since their names give a good approximation to what they do, they are not discussed on this paper.[3]

3.1 Box types and constraints

Boxes represent resources, and resources have properties. To provide for a representation of resource properties, each box has an associated type and constraint set. A box type is defined to Plan B by means of a string with the type name. A constraint set is defined by a sequence of values for each property of interest for the resource. Each value is a string. Values for different properties are separated by the "!"

[3] The `delete` system call may also be used to remove previously established links.

character. The type and constraint set are specified together using a string of the form "*a!b!c!...*". For example, the box containing a program binary to be run on a Plan B system on a PC may have the type and constraints `bin!386!wave`. In this example the type is `bin` (i.e. "binary") and `386!wave` are the constraints. Here, there are two constraint elements with values `386` and `wave`. Constraint elements are *positional* so that the first value in any constraint set gives a value for the same constraint. The convention in our system is that the first constraint refers to the architecture of the machine and the second refers to a network where the machine is attached (when the machine is attached to more than one network, the second constraint refers to the one used to reach the box).

The mechanism used by the kernel to select resources by means of constraints is a constraint resolution algorithm which is inspired by the unification mechanism of the Prolog programming language. The algorithm is as follows, although its meaning will be more clear when we discuss later how Plan B uses it to select resources.

```
/* Unifies two sets
 * of N and M constraints
 */
Unify(set1[1-n], set2[1-m]) {
  if for i = 1 to max(n,m)
    unifyval(set1[i], set2[i])!=fail
  then
    return
      set of unifyval(set1[i], set2[i])
  else
    return fail
}
/* Unifies two constraints
 */
Unifyval(v1, v2) {
  if v1 = "" or v2 = ""
    return ""
  if v1 = "\%"
    return v2
  if v2 = "\%"
    return v1
  if v1 = v2
    return v1
  return fail
}
```

The algorithm determines if two constraint sets can be unified or not. If they do, the algorithm returns a constraint that is the unified version of them, and we say that the constraints match.

We have to say that the kernel itself does not rely on which constraints are used. This means that resource providers and system users can define any set of constraints desired. If a new property is considered of interest, a new position in the constraint set can be agreed upon to represent that property.

4 Name spaces and resources

The name space glues together the trees in the forest of boxes that are of interest for the application, i.e. the set of resources of interest. Plan B refers to names that have boxes bound to them as *prefixes*; the implementation of the name space is in fact a prefix table [21]. The name space enables the automatic selection of resources by allowing multiple bindings to the same name. It also enables adaptation to environment changes by allowing automatic bindings of resources that show up in the network while the application runs. The next three sections show how this is done and how the problems discussed before are solved.

5 Handling the resource selection and redirection problems

The lack of `open` introduces an indirection that permits applications to use different instances of a given resource at different points in time, depending on resource availability. It also permits the application to be unaware of the actual location of the resource on the network, and permits changes in the name space to take immediate effect in all the applications using the names affected. For example, network connections in Plan B are handled by using boxes; the box name "/b/con/nautilus:80" represents a stream connection to the service named "80" at the machine named "nautilus". The application resolves the name each time the connection is used. By using the name space and binding appropriate resources (e.g. tcp connections or serial lines) to that name the application can stay connected despite changes in connectivity.

By using binary operations like `copy` and `link` (instead of `read` and `write`), the system is aware of which resources are to be combined. This is quite important since it permits the system to:

1. Transfer the data from the source to the destination through the best path available. Unlike traditional file copying, a `copy` interface permits data to go straight from the source to the target box without placing the client process doing the copy in the middle of the data path.

2. Select an appropriate pair of resources. This can be done only if the system knows both the source and the destination. The mechanism used in Plan B is the resolution of a set of constraints associated with each resource and is discussed later.

Regarding the first point, the optimization of the path used to transfer the data may introduce significant performance improvements. Since the implementation of `copy` asks the destination box to retrieve data from the source, caching techniques can still be applied to avoid unnecessary data transfers. When the network connection between the machine executing the operation and the rest of the system is poor, the use of `copy` can make the difference between being able to perform the copy and not being able to do it at all.

Regarding the second point, the selection of the resources involved in a system call is performed by unifying constraints on the set of boxes named in the system call arguments. System calls using a single box (e.g. `info`, which retrieves metadata for boxes) unify the constraints supplied by the user with the constraints of the boxes with the given name. The first box found for which the constraints unify is the one used. On the other hand, system calls using two boxes together (e.g. `copy`, which copies a box to another) try to find a pair of boxes such that: the first one unifies with the constraints supplied for it by the user; the second one unifies with the constraints supplied for it by the user; and the results of both unifications unify.

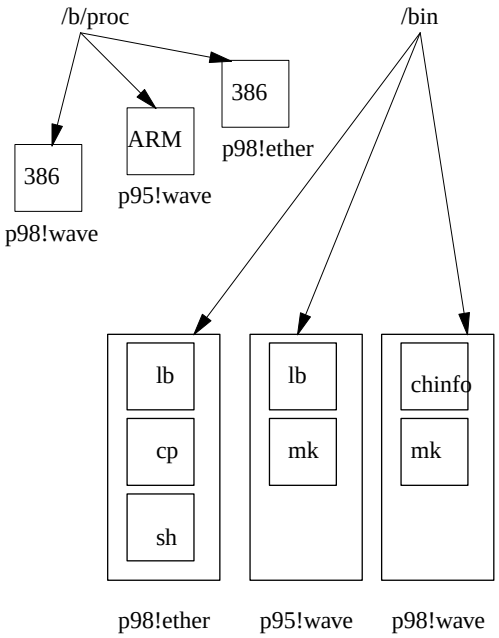

Figure 1: A network with processors and binaries

To show an example, figure 1 depicts a scenario where three different processors are bound to the name "/b/proc" and three different boxes containing inner boxes with program binaries are bound to the name "/bin". In this example, two boxes exist with name `/bin/lb`, one with constraints `p98!ether` and the other with constraints `p95!wave`. The convention is that the first constraint specifies the architecture of the machine involved and the second specifies the network where the machine is attached at (or one of them if there are many). In this example, `p98` and `p95` are two different architectures (i586/PC and sa1110/iPaq). `Ether` means that the machine is exporting the box through our fast ethernet and `wave` means that it is exported through a radio ethernet.

Now consider the system call

```
make("/b/proc/lb!\%!ether", "/bin/lb")
```

which requests the creation of a new box (a new process) named `/b/proc/lb` such that its constraints match with those of a box named `/bin/lb` and also match the constraints `%!ether`. The system searches the name space for a pair of boxes named `/b/proc` (since we are making a box, we search for a container where to create it) and `/bin/lb`. It may find first the processor box bound to `/b/proc` with constraints `p98!wave`. Such constraint set does not unify with `%!ether` and therefore the system keeps on searching until it finds the box bound to `/b/proc` with constraints `p98!ether`. Since that unifies with `%!ether` the system has a candidate for `/b/proc`. The result of the unification has the constraints `p98!ether`.

Now the system searches for boxes bound to `/bin/lb` such that its constraint set unifies with that given by the user and also with the result for other argument (`p98!ether`). Since the user said nothing regarding constraints for `/bin/lb`, the first box found whose constraints match `p98!ether` is considered a candidate for `/bin/lb`. In fact, there is a `/bin/lb` box whose constraints are exactly `p98!ether`. That is the one used. In few words, the user asked the system to create a process to execute a program and the system searched for an appropriate processor and binary program pair.

We also need some means to ask the system to choose a different resource for a given resource name and application, i.e. to perform a "resource redirection". In Plan B, this operation is `link`. Consider as an example the redirection of "standard output" for a process. In Plan B each process is represented by a box (e.g. `/b/proc/lb`) and has an inner box named `io1` (e.g. `/b/proc/lb/io1`). The convention is that this box is used by the process to serve the same purpose of UNIX's standard output. A link from a box

to `io1` would make `io1` point to such box. The next time the process copies something to `io1` data will be sent to the linked box instead. The code needed for the application to redirect the output of our example process is quite simple:

```
link(/b/term/cons,/b/proc/lb/io1)
```

Note that the process executing the call might be at a different machine and that the process involved (`/b/proc/lb`) may be already running.

`Link` also helps to keep multiple resource instances to let the system choose among them. This leads to the second meaning of `link`, namely, to maintain replicas of resources. A clear example is the design for storage boxes. A storage box is roughly the equivalent of a traditional file system on secondary storage. A storage box considers links as a means to replicate data on different storage areas, which is clearly different from the meaning of links under `/b/proc`. If the home for user "nemo" is kept both at his laptop and at his department's file server, it is reasonable to link both boxes to let the system know. If both instances of "nemo's" are bound to the name "/usr/nemo", any path starting at "/usr/nemo" can be resolved to either of the instances depending on the state of the environment. The techniques needed to keep both copies coherent have been studied by systems like Coda [18].

6 Box conversion

Boxes must be of the same type to be operated (e.g. copied) together. All Plan B name spaces include a set of conversion definitions along with the prefix table. The converter set is made of entries that specify a program that converts data of a particular type to a different type (a null program may be specified to define subtypes and constraints can be given along with the program name to constraint where should it execute). New entries can be added with the `cast` system call, and removed later with the `uncast` system call. The `/b/sys/casts` box holds a textual representation of the conversions defined, to let the user know.

When the name space searches for compatible boxes it considers the set of conversions defined, and may cause the execution of the converter program on behalf of the user.

7 Adapting to changes

The name space can be instructed to pay attention to announces or advertisements in the network stating that a particular resource is available. The system call used is `import`, usually using the network address `any` (which means that we don't care about the resource location). For example,

```
import /b/proc any /proc proc!p98 b
```

arranges for a new prefix `/b/proc` to be entered in the name space below (`b`) existing ones. Initially, there is no box bound to that prefix. However, any box advertised in the network under the name `/proc` with a constraint matching `proc!p98` will be bound to the name as soon as the advertisement is received. Plan B uses an advertisement protocol along with the *Op* protocol (which is used to operate on boxes across the network).

When it is found that an advertised bounded box is no longer accessible, it is removed from the name space. Currently, this may only happen after an operation is attempted on the box. Although it has not been tested yet, constraints could be used to provide hints about the expected lease time for a new resource to stay. Besides, the `forget` system call can be used to forget about a previously imported box.

8 Heterogeneity

Heterogeneity of architectures and networks is dealt with by combining several tools:

1. Presenting all resources as a single abstraction and using conventions to organize the name spaces. This helps to deal with resources independently of their architecture and inner structure, because at least the interface is always the same.

2. Using constraints to express restrictions and features of the resources. This helps to know which resources to import to the name space and which ones can be combined.

3. Using the converter set to automate data translations between heterogeneous formats.

For example, the standard Plan B graphics device uses the Plan 9 image format for images. When an application uses a different format for output images, a conversion can be defined to let the application use different output devices. Furthermore, should the program used to convert the image format require a fast CPU, a constraint can be defined to determine if a `/proc` box (a processor) is considered as either fast or

slow. Thus when the system tries to execute the converter, it will select a CPU considered as fast. Despite the simplicity of the constraints mechanism, this example illustrates how it can be used to deal with not so simple problems.

9 System and network failures

Plan B does not try at all to provide fault tolerance, since we believe that the user of a networked environment would experiment failures anyway. For example, a user willing to use a friend's laptop to execute a process must be aware that such process could die if the friend leaves and shuts down the laptop. In Plan B the user is responsible for calling `import`, supplying appropriate constraints, to instruct his/her name space about which kinds of resources are acceptable and which ones are not.

9.1 Network failures

Using boxes to represent network connections makes it easier to adapt to network failures. For example, a name space may have both a tcp protocol stack and an infrared protocol stack bound to the name `/b/con`, which is the conventional prefix for stream links. An application may be sending data to service 20 at machine *nautilus* by copying data to a box named `/b/con/nautilus:20`. That name is meaningful to both stacks and the first time used would issue a connection request to the remote machine. If the first network fails, the second (also bound to `/b/con`) may still take over.

One problem that may still happen is that outstanding messages going through a failing network would be lost despite the supposed reliability of the connection. In this case, the protocol used by the application over the network link is still required to provide a means to keep the connection reliable (e.g. by using serial numbers on requests and/or retrying the requests). The *Op* protocol used to perform box operations does so, which means that applications do not need to care about this issue when using boxes from the network. The implementation is greatly simplified because, if we ignore the possible error status in the result, box operations are idempotent in most cases. The lack of a "file offset" concept makes it easier to achieve this because updates are atomic and refer to an entire box, not to a part of a box.

9.2 Garbage collection

Box servers are almost stateless because box requests are self-contained. If a client crashes or gets disconnected from the system, outstanding requests are completed and the server forgets about that particular client. However, the client might have created boxes to be used just during its life (e.g. windows, temporary storage, etc.) and the server should be able to delete them when the client can no longer use them due to a crash or a network disconnection.

In this case, garbage collection is done by means of a *lease* permission bit. Clients that create boxes that should be subject of garbage collection upon client failure are supposed to set the *lease* permission bit on such boxes. Any box with that bit set would be deleted by the server if the client does not access the box during an interval of time specified by the server.

Another mechanism useful for garbage collection is the *deldie* (*delete on die*) permission bit. When set, the box involved is removed by the creator process while it exits. This helps to delete resources no longer in use when a process crashes (but the machine where it runs must be alive and connected to perform the delete operation on the box).

9.3 Consistency

The `link` operation leads to data replication when the resource implementation prefers to do so, for example, on storage boxes. This introduces the problem of maintaining data consistent between the set of replicas. Since the system is built considering that disconnections are usual, dealing with consistency is more complex.

We assume that most of the time, the user establishes links to replicate coarse grain data like a set of system binaries, a home box (what would be a home directory on other systems) and similar resources. Moreover, we assume that users are responsible for establishing a sensible set of links and that conflicts due to concurrent updates on replicated boxes are rare. With this set of assumptions, we can borrow results from systems like Coda [18] to maintain the set of replicas.

Each server holding a replica accepts updates and is responsible for propagating them to remaining replicas as soon as it is feasible. When a conflict is detected, a text message is sent to the user to notify of the conflict and permissions are adjusted to avoid further updates. When the user resolves the conflict, he/she can change permissions to allow further updates.

10 Protection

Protection is based on authenticated access checking through access control lists. Plan B checks permissions each time a box is used, which would probably not happen on a system using file descriptors. Each box has a set of permission bits (matching the operations in the *Op* protocol) to determine which operations can be performed by the box owner and by others. There are no user groups in Plan B.

The authentication protocol is left out of the system and it is assumed that box clients and servers will be authenticated before speaking *Op*. This is the approach used by Plan 9 [14]. In fact, the current implementation has a very naive authentication protocol because we plan to borrow such protocol from Plan 9.

Binary box operations like `copy` and `link`, that are performed on the destination node on behalf of the client issuing the operation, require the destination node to be able to speak for the client regarding the operation being performed. This is achieved by issuing one-shot tickets from the client to the node that performs the operation. Such tickets are used just to authenticate the user, not to perform access checking (which is done with access control lists).

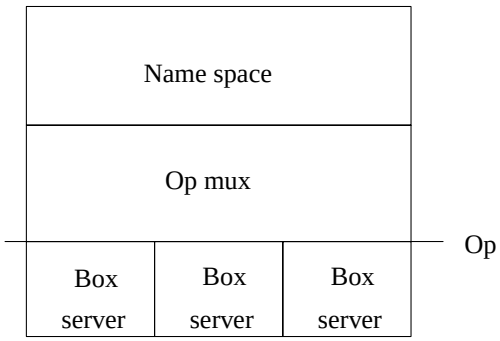

Figure 2: Overview of the Plan B architecture

11 Implementation

As of today, we have a complete implementation of Plan B, including a shell and utilities going from a "list boxes" (`lb`) program to a graphics program to set the volume on the current audio box. We rewrote the utilities from their Plan 9 counterparts to use the system in a real setting, so we could exercise the system. We hope that the examples shown in this paper will illustrate what we learned using the system. The implementation, user manual, and system description are available for download at the Plan B web site [2].

Plan B is just a box multiplexor that allows applications to build a name space for boxes in the network and tries to help them to select which of them to use on each case. The implementation follows the architecture shown in figure 2. The kernel performs each system call by resolving the box names involved and issuing *Op* RPCs to perform the appropriate operations on the boxes selected.

At the beginning of each system call, the kernel creates a `Boxref` structure for each box involved. `Boxrefs` are handles for local and remote boxes. A `boxref` carries the box name and constraints as specified by the user and keeps track of which part of the name (and constraints) are resolved and which part is yet to be resolved. During the system call, the `boxref` may point to different boxes while the name space is being searched for a box that matches the user supplied name and constraints. Once a matching box is found, the `boxref` is said to be *bound* and will not change any further. Once the system call has completed, the `boxrefs` used are destroyed. We found that this is important to make the system more reliable to changes in the environment, since it makes system calls self-contained. Although it may look inefficient, delays introduced are not appreciable for the Plan B user.

While bound, a `boxref` contains both the address of the box server and the box name as known by the server. The format used to store the address is a string *machine:service* and corresponds to a name for a network connection (*/b/con/machine:service*). Since this name may also correspond to different network links/protocols depending on the environment state, the `boxref` switches to different connections upon changes in the network (connections are established/terminated automatically by the implementation of the */b/con/* boxes).

The most complex part of the implementation is the name space, due to the complexities added by the constraint and type checking mechanisms. After a series of changes to our initial implementation of the name space, the current implementation performs searches by unifying constraints as said before in this paper. But the name space semantics was not obvious from the beginning and it required experience with the system to reach a satisfactory implementation state. For example, the constraints may refer to the box or to its container depending on the system call (e.g. `make`), and the type should be checked or not depending on the system call (once more, `make` is creating a box and therefore the type is specified to determine the type for the new box, not to do a type check).

Furthermore, since a name space lookup might be retried several times during the search for boxes with matching constraints, it is necessary to undo the unification of constraints before each attempt to find a matching box. This also happens in Prolog while performing unification of expressions, and that was precisely the reference used to reach a sensible implementation.

Binary system calls (those using two box names) resolve the names and constraints by following the code shown below, where `nsselect` resolves the prefix and `opselect` binds the handle to the box. Note that each call to `nsselect` selects a different name space entry where to try the binding. The code shown is a simplified version of the real one, which is 58 lines of C code.

```
// finds a pair that can be
// operated.
// Returns the converter to
// be used when needed.

// values for op
enum {COPY, LINK, CONVERT, MAKE};

char*
can(Boxref* sbp, Boxref* dbp, int op)
{
  nsreset(sbp);
  nsreset(dbp);
  while(nsselect(sbp)){
    if (!opselect(sbp))
       continue;
    while(nsselect(dbp)){
       if (!opselect(dbp))
          continue;
       if (!unify(sbp, dbp))
          continue;
       if (same types)
          return found;
       if (has converter)
          return converter;
    }
  }
  error(Nomatch);
}
```

Below the name space stands the *Op* multiplexor. This multiplexor is in charge of implementing `opselect` to resolve the suffix of the name not specified by the name binding (i.e. the part of the name determined by the box hierarchy in the server). Besides, the *Op* multiplexor performs the RPC call to the appropriate box server requesting the execution of the box operation. Most system calls issue RPCs not just to perform the operation, but also to learn of the types and constraints of boxes named by the user.

The current implementation works hosted on top of Plan 9. The machine dependent part of the kernel uses Plan 9 processes and files (including network connections and rio windows) to supply services to the machine independent part. The current kernel supplies boxes for processes, files, network connections, memory, graphics, sound and some other miscellaneous system boxes. A shell and several command line programs have been implemented and used to exercise the system and gain experience with its interface.

Our experience with the system shows that the performance is reasonable, although no performance tuning has been done yet. Since the merit of any performance numbers is of the underlying Plan 9 system, no measures have been made yet. A native port would be tried before. Nevertheless, measures made to the /net

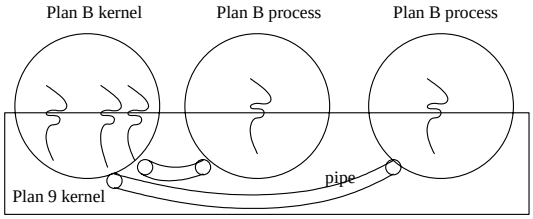

Figure 3: Implementation on top of Plan 9

framework of Plan B show that the overhead of the name space is 1μsecond that corresponds to the lookup of a name in the prefix table. This overhead remains almost constant for reasonably sized name spaces, and has been measured on a 2.4GHz Pentium 4 PC running Plan B hosted on a Plan 9 system. This is the price paid for the lack of binding. Of course, every time a box server fails, there is an overhead for the detection of the failure and the reconnection to the new server. This overhead depends on the particular network protocol used and would be the same for any other operating system willing to switch from one server to another.

The approach used to implement Plan B (see figure 3) on a host environment like Plan 9 is to employ a separate plan 9 process for the Plan B kernel, and then one Plan 9 process per each Plan B process. More precisely, the Plan 9 process used for the Plan B kernel has one thread (one Plan 9 process actually) per Plan B process. Each thread services system calls for the corresponding Plan B process. System calls are made using RPCs through a pipe between the user and kernel processes.

The source code is 3731 lines of machine dependent

code in C, and 7753 lines of portable code in the same language. Lines are counted using `wc` on all C source files under the `/src/b/9` and `/src/b/port` directories. This is just a hint of the simplicity of the system but is not to be taken too literally because we must account for the fact that Plan 9 processes and other resources are being used to implement Plan B boxes.

11.1 The Op protocol

The *Op* protocol leads to stateless box servers and is simple enough to permit implementations with tiny memory and processor usage. The protocol defines an RPC for each box operation as well as two operations called `getmem` and `putmem`. The first one is used to retrieve memory contents from a box and the second is used to update memory in the box. Unlike `read` and `write`, they transfer all the box contents since there are no "file offsets" in Plan B. When the maximum message size is not enough to perform the transfer of the box contents, a series of `getmem`/`putmem` messages is sent; the whole series is considered as a single operation. Note that by avoiding "file offsets" and updating the entire box contents at a time we reduce the problems caused by using different replicas of resources as well as those introduced by the concurrent access from clients to network resources. A failure in the middle of a series of `putmem` RPCs for a single `copy` is handled like a crash in the middle of an operation, i.e. servers should try to keep the old box contents intact until the series is completed. When this is not feasible due to the size of the box, the box will be left in an inconsistent state (very much like when the system crashes in the middle of a single `putmem`).

12 Other examples

Despite the lack of `read` and `write`, applications can still copy data from their memory to the outside world and vice-versa. In Plan B each address space is a box that has inner boxes representing portions of the application's memory. For example, the equivalent of a traditional `write` would be a `copy` from a memory box to some other box, like in

```
copy("/vm/0xf1000:0xf2000", "/some/box");
```

The `/vm` box synthesizes inner boxes on demand just for the system call that uses the inner box name.

As a final example, there is a `usr` box in Plan B that represents a human user. Among other things, it contains a set of I/O devices preferred by a user. Conventionally, the shell links its I/O devices to those preferred by the user who started the shell. Through this simple mechanism, newly started processes can*follow* the user and use appropriate I/O devices depending on the state of the human using the system. Furthermore, should the user change his mind, for example, to borrow a friend's keyboard for a while to type to an already started process, all the user has to do is to link a new set of I/O devices for the involved process.

13 Lessons learned

The constraint mechanism is very powerful when combined with `copy`. On its own, constraints seem to allow the application to select resources given a set of properties considered. However, by using the system we found most of the time that the set of properties desired for a resource depends heavily on the characteristics of another resource (e.g. we don't know which `/bin/lb` binary we want without considering which processors are available). Therefore, the combination with `copy` makes a real difference when compared with name services that simply resolve a single name given a set of constraints or properties. We hope that the examples in the paper had illustrated this, and the experience of using Plan B.

The lack of connections and file descriptors seems to make garbage collection harder. However, we found that this is not exactly true because despite connections, servers must still use either a leasing mechanism or a timeout based one to determine if a connection is just slow or the client at the other end is broken. Nevertheless, we found that this approach makes the system more resilient to failures.

Most of the times, we are more worried about performance than we should be. We always thought that the name space implementation would be the bottleneck, but experience said otherwise. Humans using the system noticed no change when a hardwired implementation that did not use constraints nor searched the name space multiple times was used instead. The real performance problem was found in the initial version of the *Op* protocol (not shown in this paper). This version issued `copy` requests to the data source and not to the data destination, which was preventing the use of data caches to avoid unnecessary data transfers. A "don't cache" permission bit was added to permit the user to request for a box to be transferred on all requests.

Building a new system by starting with a hosted version and not a native implementation is a very good approach. No OS toolkit has been used to implement Plan B, yet it is benefiting from all sort of facilities provided by a host system. We saved a huge amount

of time compared to our earlier experience of building system kernels using a native environment; even though we built them using facilities supplied by the OSkit toolkit [7].

14 Related work

Plan 9 [14] is a distributed system that is built by exporting all resources as files and allowing those files to be accessed through the network. Plan B borrows many of the Plan 9 ideas. There are some important differences though. Unlike Plan 9, Plan B uses boxes instead of files permitting data to flow without the controlling client intervention. This is important to permit clients with bad connectivity to control the transfer of huge amounts of information. Besides, along with the box abstraction comes the use of constraints (to determine box selection according to the expected usage for the resources), the lack of file descriptors (providing better tolerance of network failures), and the ability to listen for new resources in the network (to adapt to environment changes).

File systems using typed files like that of Nemesis [5] are obviously ancestors of the box abstraction. They do not consider how resources are used together in order to help with resource selection. Moreover, since their operating systems rely on files and file descriptors, they do not provide a means to perform resource redirection nor to adapt to environment changes.

This applies also to facilities like the BeOS file system [11], the Semantic File system [8] and the name service in Globe [12]. They are able to select resources that present a set of properties by means of attributes. However, they consider resources on their own, without paying attention to how are they used together. Furthermore, such systems do not optimize the paths used for data transfers and it is not clear how they permit the application to automatically adapt to changes in the network.

Some systems permit flexible access to network resources, such as Odissey [13] and Khazana [4]. Although some of them consider disconnected operation and adapt to changes in the connection status of the client machine, it is not clear how they adapt to other changes in the network, for example, changes in the links used by the servers. This is important since mobile devices are likely to be "servers" for I/O devices.

The problem of considering context to improve applications is addressed by systems like the Context Toolkit [17] and Gaia [15]. Although some of the problems raised by the willingness to exploit context information during user requests can be addressed by using constraints, Plan B is not designed as a system for active spaces.

Middleware systems like Globe [19] and WebOS [20] are targeted to solve a different problem. They are more concerned with scalability and interoperability of existing systems than they are with rethinking which services such systems could provide to make things easier.

Many mechanisms have been built (usually as middleware) to address the problems we address on this paper. Resilient overlay networks [1] and Iceberg's automatic path creation service [22] describe different means to let applications switch to different network links, which is a concrete example of what we named the resource redirection problem. The Placeless Documents framework [6] provides a means to select instances of documents in the network and automate conversions between document format; PAST [16] provides a file system that supports replication and permits selection of appropriate file replicas. Such mechanisms are either designed for a concrete kind of application, or supply one of the multiple services that an operating system is expected to provide. Plan B differs in that it provides a general purpose environment to build and execute applications.

Other systems, including Ninja [9] (whose architecture for services is called SEDA) and One.world [10] are designed to provide services by interconnecting small special-purpose devices through the internet. Although Plan B considers that there might be many small devices exporting services, it has been built as a general purpose computing environment.

15 Future work

In the near future we will port more applications from Plan 9 to Plan B, to get a complete development environment. By using the system for daily work its shortcomings and strengths will be better noticed. Later, a native port to run on Intel based PCs will follow.

References

[1] D. Andersen, H. Balakrishnan, F. Kaashoek and R. Morris. Resilient Overlay Networks. *In proceedings of the 2001 Symp. on Operating System Prin.*, 2001.

[2] Laboratorio de Sistemas. Plan B web site. At http://plan9.escet.urjc.es/who/nemo/Plan-B.html, 2001.

[3] F. J. Ballesteros and S. Arevalo. The Box: A replacement for files. *Proceedings of HotOS-VII*, IEEE Hot Topics on On Operating Systems. AZ, USA. 1999.

[4] J. Carter, A. Ranganathan and S. Susarla. Khazana. An Infrastructure for Building Distributed Services. *Proceedings of ICDCS'98*, 1998.

[5] S. Childs. Filing system interfaces to support distributed multimedia applications. *Eighth ACM SIGOPS European Workshop Support for Composing Distributed Applications*, 1998.

[6] P. Dourish, W. K. Edwards, A. LaMarca, J. Lamping, K. Petersen, M. Salisbury, D. B. Terry and J. Thornton. Extending Document Management Systems with User-Specific Active Properties. *ACM Transactions on Information Systems* 18, 2 (1999).

[7] B. Ford, G. Back, G. Benson, J. Lepreau, A. Lin and O. Shivers. The Flux OS Toolkit: A Substrate for Kernel and Language Research. *Proceedings of the 16th Symp. on Operating System Prin.*, 1997.

[8] D. K. Gifford, P. Jouvelot, M. A. Sheldon and J. W. O. Jr. Semantic File Systems. *Proceedings of the 13th Symp. on Operating System Prin.*, 1991.

[9] S. D. Gribble, M. Welsh, R. Behren, E. A. Brewer, D. E. Culler, N. Borisov, S. E. Czerwinski, R. Gummadi, J. R. Hill, A. D. Joseph, R. H. Katz, Z. M. Mao, S. Ross and B. Y. Zhao. The Ninja architecture for robust Internet-scale systems and services, Computer Networks. *Special issue on Pervasive Computing.* 35, 4 (2000), .

[10] R. Grimm and B. Bershad. Future directions: System Support for Pervasive Applications. *Proceedings of FuDiCo 2002*, June 2002.

[11] B. Inc. *The Be Book*. California USA. 1997.

[12] I. Kuz, M. Steen and H. J. Sips. The Globe Infrastructure Directory Service. *Computer Communications* 25, 9 (June 2002), .

[13] B. Noble, M. Satyanarayanan, D. Narayanan, T. J.E., J. Flinn and K. Walker. Agile Application-Aware Adaptation for Mobility. *Proceedings of the 16th ACM Symp. on Operating System Prin.*, 1997.

[14] R. Pike, D. Presotto, K. Thompson and H. Trickey. Plan 9 from Bell Labs. *EUUG Newsletter* 10, 3 (Autumn 1990), 2-11.

[15] M. Roman, C. K. Hess, R. Cerqueira, K. Narhstedt and R. H. Campbell. Gaia: A middleware infrastructure to enable active spaces *Technical Report* UIUCDCS-R-2002-2265. University of Illinois at Urbana-Champaign, 2002.

[16] A. Rowston and P. Druschel. Storage Management and caching in PAST. A large-scale persistent peer-to-peer storage utility. *Symp. on Operating System Prin.*, 2001.

[17] D. Salber, A. K. Dey and G. D. Abowd The Context Toolkit: Aiding the Development of Context-Enabled Applications. *Proceedings of CHI'99.* ACM Press., 1999.

[18] M. Satyanarayanan. Scalable, Secure, and Highly Available Distributed File Access. *IEEE Computer* 23, 5 (May 1990).

[19] M. Steen, P. Homburg and A. S. Tanenbaum Globe: A Wide-Area Distributed System. *IEEE Concurrency*, Jan-Mar 1999.

[20] A. Vahdat, T. Anderson, M. Dahlin, D. Culler, E. Belani, P. Eastham and C. Yoshikawa WebOS: Operating System Services For Wide Area Applications *Proceedings of the Seventh Symposium on High Performance Distributed Computing*, 1998.

[21] B. B. Welsh and J. K. Ousterhout Prefix tables: A Simple Mechanism for Locating Files in a Distributed System *Proceedings of the 6th ICDCS*, 1986.

[22] Iceberg Project. Automatic Path Creation Service, http://iceberg.cs.berkeley.edu/release/APC.html.