A New Network Abstraction for Mobile and Ubiquitous Computing Environments in the Plan B Operating System

Francisco J. Ballesteros Eva M. Castro Gorka Guardiola Muzquiz Katia Leal Algara
Pedro de las Heras Quirós
Laboratorio de Sistemas — Universidad Rey Juan Carlos
Madrid — Spain
{nemo,eva,paurea,kleal,pheras}@lsub.org

Abstract

Today, there are several different communication interfaces available and we plug/unplug them at will. However, our network programming interfaces remain similar to those used when there was at most one network interface.

As a result, applications must struggle with the problem of adapting to changes in network facilities: they must select the appropriate network, switch to another when the current one fails, re-establish failed connections, and so on. Although there are systems and middleware that solve part of these problems, there is no integrated approach that addresses the underlying problem: a new network programming interface is needed for applications that must run in mobile and ubiquitous computing environments. This paper presents the design, implementation, and experience with a novel network programming interface introduced in the Plan B's *Operating System and called* ***/net****. It also shows how this addresses the problems mentioned and simplifies network programming, including some example applications.*

1. Introduction

Usually, we expect the operating system (or the middleware) to do both resource management and resource abstraction on behalf of the application. However, in mobile and ubiquitous environments, the system underlying the application (be it an OS or middleware) does not has much support for network programming. Mobile devices usually have multiple communication means available (networks using TCP/IP over ethernet or 802.11, as well as serial, infrared, and/or bluetooth connections). When an application wants to communicate with a remote program that is only reachable through some of the networks available, it must still know which network to use.

Although we might eventually expect the system to take over that task, nowadays, the application has to do the system's job. For example, the network used may be replaced by another, which might be using a different protocol family (note that it could be just an SMS facility!). In this case, it is the application that has to detect the communication problem, discover that a different network is available, and reroute its data messages through the new network. Once more, it is doing the work that the underlying system should be doing. The problem is that the network APIs we use were designed when it was not common to have multiple networks available, and it was even less common to consider that they might become enabled and disabled during the application's execution.

For example, a program sending a print request cares about the destination printer service used, not about which network is used to reach the printer. Similarly, a remote execution facility wants just to send commands to the appropriate machine, an editor cares about reaching a file service to save or load a file, a web navigator cares about reaching a server, and the list goes on. This does not mean that a program should not be *able* to specify which network to use; it means just that the program should not *have* to. The application might require the network to have certain properties like a reliable transport, high bandwidth, or the like. But note that needing a network with a particular property is not the same as needing a particular network.

In this paper we introduce a novel network abstraction, called /net, that allows applications to specify endpoints in the network(s), without specifying which network should be used each time communication happens. Our API also allows networks to be selected based on their properties, making it easier for applications that care about which transport is used. Our system relies heavily on the traditional file abstraction, per-application name spaces, and the use of (volatile) late binding. By doing so, we permit the system to switch from one network to another depending on their availability, without placing the burden on the appli-

cation. Because imposed transparency may be harmful for the application, we give the application the power to control the extent to which the system will adapt to networking changes. This control can be exercised to request complete automatic adaptation, complete manual operation (similar to programming in other systems), or some behavior in between.

Our approach has been to replace the network abstraction supplied by the system in a way that would be trivial to port to other operating systems. This way, legacy applications and systems can also benefit from adaptation facilities, and there is no need to adhere to any particular middleware. Furthermore, it took less than one hour to port `/net` to the Plan 9 OS, and we do not expect that porting it to Windows or UNIX should be much more difficult.

The rest of the paper is structured as follows: Section 2 shows the Plan B OS network API, /net, and how it helps with adaptation. Section 3 further describes how applications adapt with /net, introducing some examples of use. Section 4 discusses some problems we found. Section 5 gives some measures and implementation details. Section 6 discusses related work. Finally, section 7 concludes and describes our plans for future work.

2. The Plan B's network API: /net

Plan B [4] is an operating system built specifically for use in highly distributed and dynamic environments, like those available in mobile computing and in ubiquitous systems where resources can be added and removed dynamically. It has been designed to interoperate with existing systems and can be used both on the bare machine or hosted on top of an existing system. Its abstractions are very high-level, to tolerate implementations on a wide range of platforms going from a native hardware platform, to a Windows, UNIX, or Plan 9 machine. There are a few assumptions underlying Plan B that are important to the design of its network interface:

- **There are many alternative resources to choose among.** For each resource wanted, there are many alternatives available. For example, there may be many network endpoints (reachable through different networks) that might correspond to the service named `/net/myserver:http`.
- **Resources are highly dynamic.** The set of resources available for a given name greatly vary over time. For example, the system is likely to switch from one network to another while an application is using the name above.

These assumptions lead to the following principles that heavily influenced `/net`:

- **We use volatile late binding.** Names bind to resources just to perform each operation requested by the application, and the binding is removed as soon as the system call completes. In Plan B, applications use names (and not handles, descriptors, proxies, or similar artifacts) to perform system calls.
- **Each application has a name space.** Because names determine which resources are used, and because different applications may have different needs, each application has its own name space and can customize it. The name space is implemented by an ordered prefix table [20] that specifies name prefixes that map to the different servers where resources are found.

2.1. Names as tools for adaptation

The first tool provided by the system to the application is the construction of the per-application name space. As seen in figure 1, resources are provided by servers (e.g., protocol stacks) that give names to them (e.g., `nautilus:http`). The name of a resource is made of the prefix installed in the name space, followed by the (suffix) name used by the server implementing the resource. When the application makes a call using a resource name, the first prefix table entry that matches the name and has a server servicing the call is the one used.

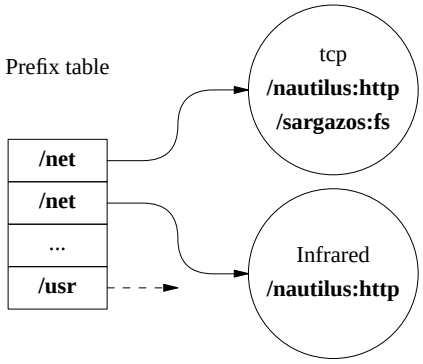

Figure 1. Plan B resources and names.

Each application (or an ancestor) may create or change its name space by defining names where servers are imported (e.g., `/net`). We provide two calls for this purpose: `import` and `forget`, roughly analogous to UNIX's `mount` and `umount`. `Forget` undoes the effect of `Import` and is not discussed here.

`Import` gives the application the necessary control over adaptation. If an application requires using TCP, it may import to `/net` just the TCP/IP stack. When that application creates an endpoint, (e.g., `/net/nautilus:http`), that

endpoint will be serviced by TCP. Should TCP fail, the application will have to cope with that. There is no adaptation to environment changes in this case.

On the other extreme of the spectrum, an application that wants fully automated adaptation, and does not care at all about which network is used, may import to `/net` all the known protocol stacks. In this case, the system chooses (on each call) which network to use. Should one network fail, the system would choose another that works and use it to try to reach the peer. For connection oriented transports, the network is considered to work if a connection can be stablished with the peer. For other transports, the network is assumed to work if it does not report any error while creating the endpoint or sending to the peer. Note that since the customization of the name space may be performed by an ancestor of the application, the application does not need to perform the imports itself.

When automating adaptation to changes, it is important not to loose properties of interest to the application. For example, an application may require a reliable and high bandwidth transport, no matter which network is used. The `import` operation allows the specification of properties that must be met by the resources imported to the given name, as [4] and the Plan B user's manual in [7] show in more detail. For example, this command or its equivalent system call

```
import /net any /net!HByes!Ryes a
```

would import to the name (prefix) `/net` any protocol stack (resources known as `/net`) that has `yes` as the value for high bandwidth (`HB`) and reliable transport (`R`) properties. For each operation performed later to endpoints created at `/net`, the system will choose which network to use. The final parameter in the command shown above determines if the resource is added before or after the ones already installed for that name.

Per-application name spaces are a powerful mechanism because this permits customizing the environment at any time. As an example, an application may perform the import shown above in the hope of using an application-level protocol designed for good connectivity. If all transports labeled as "high bandwidth" fail, the application will receive an error the next time an endpoint is used. It might then switch to a "low-bandwidth" mode of operation and change the name space to request just reliable transports (with no further requirements). Our abstraction enables the strategies used by systems like Odyssey [11] where the application operates in different ways depending on the environment. Note that, in our system, the application would not be disturbed if an alternate transport with the desired properties is available.

2.2. Services, endpoints, and addresses.

The second tool provided by /net to help with adaptation consists of using a single naming scheme and a single programming interface for all the networks. All operations performed at `/net` are handled by the implementer of a network service. A network service supplies an abstract resource called a *network endpoint* (actually, it supplies many of them.) A network endpoint represents a point in the network where data can be sent to or received from. The only interface for this abstraction is made out of these operations: `make` creates an endpoint, `delete` shuts down an endpoint, and `copy` copies data to or from an endpoint.

To permit the overlay of networks, our abstraction forces all endpoint names to be independent of the network used. Unlike other systems, the application does not have to resolve the name. It supplies the name and the network service is responsible for resolving it. Note that the overlaid networks do not need to be homogeneous (e.g., IP-based). Our approach can be used to overlay heterogeneous networks as well, because the interface is high-level enough to remain independent off a particular network technology.

An endpoint name consists of a machine name, a service name, and a number. The number is used to disambiguate names that have the same machine and service names, and is usually a random number. The translation between machine/service names and network addresses is done by each network service, not the application. This means that each network in our system includes a name service to perform the translation, but we must emphasize that such service is independent from the application.

The mapping between abstract names and concrete names is kept in a per-network database with an entry for each machine or entity. In order for this to work, we must adhere to strict conventions about how to populate this database. For known machines or machines with known IP-based networks, the names are those translated by DNS. For well knows services, we use an standard name (e.g., "http").

For unknown machines or those without (known) IP-based networks, the mapping is network dependant, but follows conventions for uniqueness and legibility. For example, for bluetooth, which requires explicit pairing by the user, we fill the database with a dns-like name. For services without names, but with a port number, we use the port number.

There are several kinds of endpoints, depending on the machine part of their names:

- `local:`*`svc`*`:`*`N`* corresponds to a, so called, local endpoint. It represents endpoints created in the local machine to accept new data from other places in the network targeted to the named service *svc*. Once a peer has sent data to a local endpoint, such endpoint is

bound to that peer. Therefore, data can be copied to a local endpoint for replying to a request received from it. To allow multiple peers to send data to a local endpoint, a process can create multiple local endpoints with the same service name and different values of N. Our convention is to use integer values for N, but any other name can be used.

- `machine:svc:N` is the name for an endpoint created to send data to the named service running at the named machine, and to receive further data from it. Once data has been sent through it, the endpoint is bound for the peer, which means that copying from it can be used to receive replies to requests sent.
- `remote:svc:N` is not and endpoint, but an artifact used to learn the address of a peer. It contains the name for the peer endpoint of the local one with the same *svc* and *N*. The peer name is supplied following the same naming convention that is used to name endpoints, (i.e., *machine*:*svc*:*N*). Therefore, the name read from a `remote` endpoint can be also used as an endpoint name.
- `all:svc:N` corresponds to an endpoint used to send data to all machines where the named service runs (i.e., to perform broadcasts).

2.3. Using /net without Plan B

What has been said considers `/net` within a Plan B machine. Most of the benefits of `/net` come from the ideas underlying Plan B that we described previously in this paper and elsewhere [4]. The network framework presented in this paper comes from aplying the Plan B's approach to a network programming interface.

Nevertheless, using `/net` without Plan B is feasible and easy to achieve. The only requirement is to include as part of the `/net` framework the implementation of a Plan B name space. This implementation consists of two small C files (588 and 308 lines) that implement the prefix table for a name space.

To support this claim, we argue that the code runs hosted on top of Plan 9 and has only a few implementation dependencies. We expect that it would be very easy to place it in a library. The code for the `/net` framework itself is 726 lines of portable C code plus some platform dependent code. For the Plan B implementation hosted on top of Plan 9, the platform dependent code accounts for 423 lines (also in the C programming language).

3. Using the network and adapting to changes

The code used by a web navigator to retrieve a web page from `nautilus` is shown below. Note that, before executing it, the networks to be considered must be instantiated by importing them to `/net`. We show later an example of how to do it.

```
// Example code for a web navigator
char req[], repl[]; // data

// 1. create the endpoint
make("/net/nautilus:http:0");
// 2. send the request
copy("/net/nautilus:http:0", 0,
   "/mem", &req, sizeof(req));
// 3. get the reply (n bytes)
n = copy("/mem", &repl,
     "/net/nautilus:http:0", 0,
     sizeof(repl));
// 4. hang up.
delete("/net/nautilus:http:0");
```

In (1), the program calls `make` to create an endpoint to talk to the web service at nautilus. What network is actually used depends on which ones are imported to `/net` in the name space. As far as the program is concerned, the endpoint is created in the first network available that works. Statements (2) and (3) copy data to the endpoint (sending the request) and then from it (receiving the reply). We have to say that `/mem` is a Plan B resource that represents *all* the application's memory, offsets and lengths have been supplied: the first call is similar to a traditional write, and the second is similar to a read. Also, note that statement (3) would have to be repeated because the server reply might not be contained in a single reply message. But it is worth considering that both calls use names and that no name nor operation is specific to a particular network. The interface is portable across network services, which is very important to enable adaptation.

Adaptation happens while the application uses the network. For example, if the networks imported are those of figure 1, and both networks are initially available, the client code shown before would make the endpoint in the first (preferred) network. Should it get out of service, the network signals an error (probably during a `copy` call). Because of the signaled error, the system retries the call in the next network available that is found in the name space. If any of the networks is connection oriented, the endpoint (on its own) establishes the connection when needed to send data. The next section discusses the data loss and duplication issues that can arise due to a network stream reconnection.

For a web navigator, the imports performed might be:

```
import /net any /net!Ryes b
import /net any /net a
```

This means that reliable transports would be attempted first because they are imported before. Should none of them work, any other transport would be considered, since all of

them are imported after the previous ones. The worst that can happen is that the user might need to hit reload if an http request is lost by an unreliable transport. This is an example of an application that prefers to use complete automated adaptation.

A network file server client program, or an editor capable of saving files through the network, should use this instead:

```
import /net any /net!Ryes b
```

This means that no unreliable transport would be ever used for this application. But the system might still switch between reliable ones.

As a further example, a better network file system client (in the line of Odyssey [11]), would first

```
import /net any /net!Ryes!HByes b
```

and use a mode of operation for well-connected clients (reliable, high bandwidth networks). Then, upon a network error, execute this

```
import /net any /net!Ryes a
```

and switch to a poorly-connected machine mode of operation.

3.1. Listening for calls

To complete the description of our interface, a program servicing HTTP requests would probably spawn several processes to attend several clients. Each process would execute this code:

```
// service local:http:0
char addr[Maxaddr];
for(;;){
  // 1. make local endpoint
  make("/net/local:http:0");
  // 2. get the request
   n = copy("/mem", &req,
         "/net/local:http:0", 0,
         sizeof(req));
   // 3. log the peer's address
   copy("/mem", &addr,
      "/net/remote:http:0", sizeof(addr));
   print("peer is \%s\n", addr);
   // process the request
   do_req(req, repl);
   // 4. send the reply
   copy("/net/local:http:0", 0,
      "/mem", &repl, sizeof(repl));
   // 5. hang up
   delete("/net/local:http:0");
}
```

In (1), a local end-point is created. Once created, the endpoint is ready to accept data from the network targeted to the local `http` service that we are implementing. In (3) we show `remote:http:0` is the means to retrieve the address of the peer (if supplied by the protocol). Another thing to note is that deleting the endpoint would hangup to the peer. Again, statement (2) should be repeated until a complete request has been received. The code of a complete http server built along these lines can be obtained from the web site `http://b.lsub.org/ls/planb.html`, serviced by a Plan B system that uses the interface shown in this paper. Such code is not more complex than the one shown here, yet it adapts to changes in network availability.

For programs sending requests, the mechanisms described so far suffice to adapt. However, for programs listening for requests, a mechanism is needed to listen for calls on all networks at the same time. The problem is that a make of a local endpoint (step 1 in the code above) would create that endpoint only at the first network found in the application prefix table that happens to service the call. This would make the server unresponsive for other networks that are not the preferred one.

The mechanism we have used to fix this problem is the introduction of a `mux` network that does not correspond to any protocol stack. Instead, `mux` is a multiplexor that services local endpoints by using the other networks. When a local endpoint is created in `mux`, it creates one local endpoint on each one of the other networks. When a copy is performed from the local endpoint, it is serviced by trying to copy from any of the endpoints in the multiplexed networks. That way, a server that wants to listen for requests coming from any network available, only has to import the multiplexor network and create its local endpoint there. The multiplexor then takes care of listening on all the networks available. Apart from the issue of listening for incoming calls, the mechanism provided by the names and the interfaces suffices to let servers adapt to the networking environment like clients do.

3.2. Adaptation problems

As shown in the previous section, the first problem we identified was that switching among heterogeneous networks required the introduction of a multiplexor to listen for calls from all networks at the same time. When networks are homogeneous, the underlying protocol could still manage to keep the server listening. For example, when switching between TCP through ethernet and TCP through 802.11, the `listen` call used by our implementation kept the server listening at the given TCP port for all known IP addresses. Thus, when we switched off one network, the other was already listening. However, this does not happen when switching from TCP to a completely different protocol (e.g., bluetooth, IR, etc.). For this case, it is necessary to

implement a listening endpoint as a series of endpoints: one per network, as described in the previous section.

Another problem is that the timeouts used by the underlying protocols to detect that a network has failed can be annoying. For example, the longest backoff time for TCP in the protocol stack we use is 30 seconds. A long timeout is good if we only have one tcp network and are not willing to hang up our connections easily, before waiting for the network to recover from a transient failure. However, if our plan is to try any other available network instead, such a long timeout does not work well. In Plan B we have shortened all such timeouts, because they heavily affect to the overall failover time and should not be bigger than one second for interactive applications.[1] An alternative was to implement a separate mechanism to detect network failures and leave the underlying protocol timeouts unchanged, but given the network hardware in use today, it is better to change the timeouts than to provide an alternate detection mechanism.

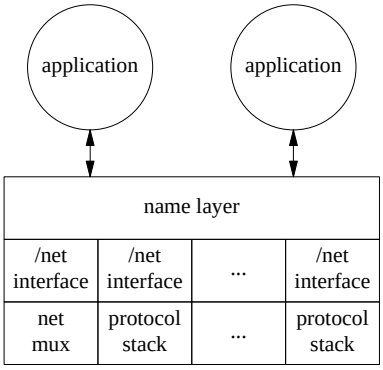

Figure 2. Architecture of /net.

A well known problem is how failover affects the data still in-transit through the network (i.e. data kept in the buffers of the sending/receiving protocol stacks during a network failover.) Some systems that permit reconnection of TCP streams combine buffers and acknowledgements to ensure that lost data is retransmitted. For example, Rocks [21] retransmits data when needed after switching from one TCP connection to another one. Our approach does not need that additional mechanism to ensure that data has been sent. In Plan B, when a network signals a failure, the failed operation is retried through another network. This means that the buffering done by the implementation suffices to perform retransmissions through the new network (like it does traditionally to retransmit through the same network). Another problem of this approach is that it can lead to duplicate data under certain circumstances, but that is a different issue.

Failover may lead to duplicate data being sent. This happens when a network sends the application data and afterwards it signals an error. In this case, the system would pick another network to send the data through. The result is either a duplicate request or reply being sent. To solve this problem, we can apply the approach used in Rocks, which uses a sequencing mechanism to discard duplicate data in the event of a failover. Although Rocks can failover only for homogeneous networks (e.g., switching from TCP/IP to TCP/IP), its sequencing mechanism can also work for heterogeneous ones. We do not include this mechanism in our current implementation. Most of the network communication in Plan B is performed to access remote resources through a protocol called BP [4] that already includes sequence numbers and addresses this problem too.

4. Implementation

The implementation of Plan B's /net follows the scheme depicted in figure 2. Applications talk to the system through the name layer. This layer implements the prefix table and is part of the name space implementation of Plan B. When a call is made, the name layer scans through the application prefix table to locate any entry that matches the name of the resource used in the call.

Once a matching entry is found, the operation is tried in the corresponding server (see figure 1). Should the operation fail there, the operation is retried in the following matching entry of the prefix table, if any. This means that the naming layer is overlaying the set of resources available.

Usually, applications import to the prefix `/net` either the network multiplexor or some set of networks. Therefore, for calls that refer to names under `/net`, either the network multiplexor or a protocol stack would handle the call. Note that all the networks and the network mux share the same interface. Namely, the interface provides `make`, `delete`, `get`, and `put` calls (The last two ones are used to implement `copy`). As it was said, the multiplexor is simply a convenience, to service "listen" requests for multiple networks. It performs `make` requests for local endpoints on each network when a mux receives a request to create a local endpoint on it.

The network interface is *abstracting* the underlying protocol in a portable manner by supplying the operations mentioned above. Figure 3 shows this. An application would use endpoints implemented by the network interface modules of figure 2 that map to concrete network endpoints supplied by a particular network or protocol stack. In figure 3,

1 It would be better to let the users select how tolerant they are with problems in their networks. Some might be willing to wait more and avoid some reconnections; others might be eager to declare a network offline and avoid waiting before switching to another one. We plan to explore this issue in the future.

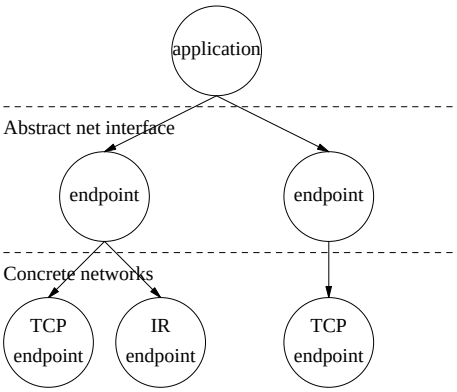

Figure 3. Endpoints and /net.

the abstract endpoint on the left may be one provided by the multiplexor, which created two different endpoints to listen at two different networks; the one on the right is the typical endpoint created by any other network interface. How this abstraction works is described in the following sections where we show how it is implemented. We do so by discussing how each operation is performed for each kind of endpoint.

4.1. Local endpoints

When a `local` endpoint is created, due to a call to `make`, the network interface binds a port to the service and announces it. The port used depends on the name used by the application (that is translated by an entry in the `db` resource). A further `copy` from the endpoint is implemented by listening for an incoming call and accepting it in the case of connection-oriented networks. In any case, the implementation does a receive in the protocol stack involved and services that data to the copy operation. Note that since `local` endpoints are created to accept incoming requests, the implementation does not admit a copy to a `local` endpoint before some copy has been performed from it.

After the first data have been received, the network interface implementer creates a (fake) `remote` endpoint matching the name of the local endpoint used, and places into it the address of the remote peer. This is to let the application know the address of its peer, should it care.

The implementation keeps the local endpoint bound to its peer until the connection breaks (for connection-oriented networks); or until a further copy operation is performed to receive more data (for datagram networks).

While the endpoint is bound, further copies to or from it are implemented by sending or receiving data from the corresponding endpoint in the underlying network. The remote address where data is sent to (or received from) is that of the bound peer. When the connection breaks (or when it is a datagram network), the implementation makes a single attempt to reconnect to the remote peer. Should this attempt fail, an error condition is signaled (which usually makes the system switch to a different network). This retry-once mechanism is useful for reestablishing broken connections on working networks and also allows the system to detect non-available networks (those that fail multiple times) in a reasonable time (this would trigger adaptation by switching to another network).

The implementation of a delete operation is as easy as closing the connection, if any, and destroying any underlying resource used for the destroyed endpoint.

4.2. Non-local endpoints

Non-local endpoints are implemented like the ones described above. However, the implementation expects an initial copy of data to the endpoint before copying data from it. Once some data has been sent, the network implementation has an established connection to the peer, and the application is allowed to copy (receive) data from it.

Connection break handling and the retry-once mechanism work as shown in the previous section.

4.3. Broadcast endpoints

Broadcast endpoints only admit copy of data to them (not from them). The implementation uses the broadcast primitive of the underlying network to send the data. If the underlying network does not support broadcasting, a creation of a broadcast endpoint fails, and the system retries the operation in another network. For the user, the operation works as long as one network imported to the name `/net` supports broadcasting.

4.4. Raw endpoints

What has been described suffices for 99% of the applications. For the 1% that requires a complete control over an underlying network, each network provides a *raw endpoint*. A raw endpoint has the name `raw:none:N` and is a hook right to the underlying network. The application is expected to copy requests into this endpoint and then copy replies out of this endpoint. The set of requests is dependent on the network used, and is similar to a traditional interface like [14].

We have to say that none of the applications we use for our daily work would require the usage of this kind of endpoint if ported to Plan B. None of the ones already written or ported make use of it.

5. Evaluation and experience

In this section we show the evaluation made for the system and outline the experience we gained by using it.

5.1. Performance evaluation

We introduce several sources of overhead with our approach that we measure in this section. All measurements mentioned below have been performed on a 2.4GHz Pentium 4 PC running Plan B hosted on a Plan 9 system. To avoid interference from the networks, we have used loopback ones. All figures below correspond to the average of 100 measures taken while the machine was idle. The clock used to obtain the time is based on the time stamp counter register of the processor. This register counts at the processor speed. The kernel used this as its source of time. We read the counter from user code by reading its register. The overhead of reading a register is negligible with respect to the measurements.

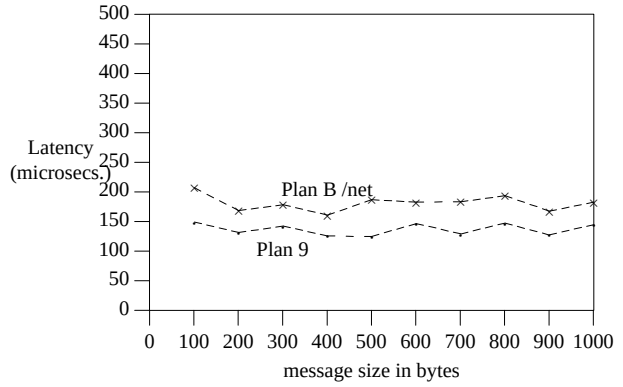

Figure 4. Message latency for Plan B /net and Plan 9, for different message sizes.

It is not feasible to compare measures made to our implementation with measures made to a traditional socket interface, because that would require either implementing `/net` on a traditional system or implementing a traditional network abstraction in Plan B. Instead, we obtain an upper-bound on the overhead of using `/net` by comparing our system while running hosted on Plan 9 with the underlying Plan 9 system. The measured overhead includes both the overhead of `/net` and the overhead of using a hosted implementation. The measured throughput for such setup is the same for our approach and Plan 9; our overhead is seen in the observed latency. Figure 4 shows the latency (in microseconds) for sending different message sizes (up to 1000 bytes). The solid line corresponds to a Plan 9 program that used the Plan 9 network API. The dashed line corresponds to the analog Plan B program, running on a Plan B hosted on the same Plan 9 platform. On average, the latency of `/net` is 44.2 μs worse than the Plan 9 one. But as we said, this is a very conservative upper bound for our overhead, because this time includes many more crossings of the user-kernel boundary than a native version would have.

To provide a optimistic lower-bound for the overhead of `/net`, we considered just the overhead of resolving names each time the interface is used (instead of permanently binding names to resources). To measure this, we executed the `info("/net")` system call, that simply resolves the name and returns metadata for the named object. The result is 28.7μs in our machine for the complete system call, and 27.7 μs if we remove the part that resolves the name, which means that it takes 1μs (in average) to resolve a name in the prefix table. In a system that has binding (i.e., on systems other than Plan B) this 1μs can be avoided. We thought this time would heavily depend on the number of entries installed in the prefix table, however, measurements show that the actual time spent in the search is negligible with respect to other factors (such as the length of the name being resolved, or the number of interrupts that the machine serviced while searching the table.) This makes sense if we think that each prefix found before the one actually used implies just another string compare operation. Fortunately, this overhead is not significative, if we compare it with the time employed by the underlying protocol stack and the network device.

What has been said applies to an operation in the absence of failures. When a network fails, there are sources of overhead, discussed below, that are unavoidable and have to do with the tasks needed to perform a reconnection.

The time needed to perform a failover to a different network depends on both the time needed to detect that the current network fails and the time needed to switch to a different one. Both times are specific of the underlying protocol stacks and are not introduced by our approach. The former is considerably larger because it involves timing out a failing network. The call that happens to use a failing network for the first time suffers the whole delay. Subsequent calls recognize the network as failed and take just the time to switch to the new network. The failure detection time can be on the order of seconds and is imposed by the underlying network protocol. In Plan B the delay is two times bigger, considering the retry-once mechanism.[2]

The time to switch networks depends on the network we are switching to. This time corresponds exactly to what it takes to initialize a new endpoint for use in the new network and is also unavoidable. Its value is what it takes for the par-

2 We plan to let our users select between being tolerant with transient failures or not. A user adjustable parameter would simply impose a maximum on the timeouts used by the protocol stacks to tolerate transient disconnections.

ticular protocol stack to create its end-point, and makes negligible the time to allocate the abstract end-point data structure. In the hosted Plan B we have used to measure performance, `/net` takes 1.337 μs to create an endpoint (this includes creating it, binding an address it, and listening on it); The underlying Plan 9 platform takes 0.678μs to do the same operation.

We have measured the time to retrieve a web page using the Linux HTTP client `wget` against a server running on Linux, Plan 9, and Plan B. The machines involved were 2.4GHz Pentium 4 PCs connected through 100Mbps Ethernet. On average, it takes 22ms for Linux, 26ms for Plan 9, and 26ms for Plan B. Although this measure accounts not just for /net, but also for the differences in the ethernet drivers, the TCP/IP stacks, and the OS kernel, it suggests that /net is competitive with respect to other systems.

Taking into account the measured overhead and the benefits of our approach (both simplicity and the ability to adapt) we consider that it is worth using `/net` for network programming in dynamic environments.

5.2. Experience

We have used /net to access services using a protocol similar to those of network file systems, called BP [4]. In addition, we have been using applications to exercise the interface, including a web server, as well as remote access for audio devices and their volume controls. We have found that the interface is extremely simple to use while making our applications much more resilient to changes in network availability. Furthermore, applications that do not include even a single line of code for adaptation to network changes, become adaptive. The import mechanism allows us to tune the adaptation for them, as we saw before.

The final interface presented in this paper may look obvious, but it is not. We made many mistakes that we found only after using the interface. Most notably, in our first version, we did not provide a means to hold multiple connections to the same remote endpoint, and had to introduce the final number in the endpoint names to disambiguate). Also, we did not foresee that a multiplexor was needed. The way to map abstract endpoints to underlying connections, and to learn peer addresses, evolved heavily as we were using the API, before reaching the model presented in this paper.

6. Related work

Both Plan B and its /net framework are heavily influenced by Plan 9 [13] and its network interface [14]. Plan 9 uses a portable file-like interface as its network interface, which is considerably simpler than other APIs including sockets [18] and Java Streams [10]. Our interface is no more complex than the Plan 9 one (and may be simpler!), and unlike the systems mentioned, we are able to switch to different networks as the environment changes. Unlike our system, both Plan 9 and the systems we mention below keep file descriptors, socket or streams descriptors, or object references, after the initial setup of their network endpoints. We use names and do not use descriptors. Therefore, the systems mentioned either do not adapt or have to introduce more software to re-introduce an indirection between the application and the network. They lost such indirection when they decided to use descriptors.

Some researchers have suggested to introduce more binding levels to help in issues like routing and location of network destinations [16]. Our proposal is just the opposite, to almost remove binding.

Many middleware systems like One.World [5], Linda [2], Jini [19], and INS [1], include services based on tuples or attributes that permit the application to locate resources in the network in a portable way, which might permit adaptation. Other systems like Globe [8, 6] rely on distributed objects or introduce proxies to admit adaptation. But it is unclear how network programming could be performed with these systems for conventional applications that are not programmed for their middleware. Consider for example typical servers and clients for web, mail, distributed editing, and so forth. Our approach differs in that we do provide a system-level interface to do network programming while still making it straightforward to adapt.

In fact, the biggest difference between our approach and most related work is that we provide a generic mechanism to adapt (dynamic name prefixes for resources modeled like files, together with volatile late binding) that is applied to the network endpoints as well as to any other resource in our system. It just happens that our generic mechanism, designed for adaptation to changes, also addresses the concrete problem of adapting to changes in networking.

It might seem that Speakeasy [22] is related work, but it addresses a different problem and is unrelated to our work. We focus on a new network programming interface and its underlying abstraction. Speakeasy and other related technologies are something that could be implemented by using our abstraction. They are more potential users of our work than direct competitors.

DHT's [15] are being used as a complementary abstraction for network addressing, resource location, and remote access to resources, but it is not clear how this mechanism could provide a network programming interface like our approach does.

There are many works about overlaying networks (e.g., RON [3] and Rocks [21]) to help re-route connections transparently through homogeneous networks. Unlike such systems, we provide an integrated approach that also works for heterogeneous networks. Systems like Vertical Overlay Networks [17] use several networks at once, trying to use

the one that works best at any time. This is similar to /net in that it would adapt to changes in network availability. It is different in that the networks considered are still homogeneous. Also, their mechanism applies only to networks, and ours is also used to adapt to changes in other resources [4]. All that has been said applies as well to Rocks [21], MSOCKS [9], and MobileSockets [12], that add an extra layer of indirection between the application and its network connections. They switch only between homogeneous networks. They do not handle issues like naming and heterogeneity of interfaces that are necessary to map between the application view of the world and the addresses and names used by the different networks.

Another difference between our work and the systems mentioned is that our approach can be used both when the users can afford switching to a different operating system, and when the users do not want even a single modification of their operating system kernel. This is due to the fact that Plan B can work either as a native system or as a hosted system running on top of another.

7. Conclusion and future work

The contributions of this paper are the introduction of a new abstraction for network programming, the description of its implementation, a description of a generic approach to adapt to changes in the network, and a description of some example code to user our system.

Plan B /net uses a volatile late binding mechanism, coupled with the use of a high-level interface that is provided at the system level for the resource considered (the network in this paper). It provides adaptability and can switch from one network to another without disturbing the application. Per-application prefix tables permit control over adaptation so that the transparency provided may be controlled instead of getting in the way of the application and becoming harmful.

It would be useful to gather statistics about networks and to be able to use them to choose the network to be used.For example, it would be interesting to be able to use the cheapest network, or the more reliable one, or the one with a highest available bandwidth, etc.

We are currently working on the second edition of the Plan B operating system. We have an operational system hosted on top of Plan 9, and a native port for Intel PCs is underway. In the future we plan to port more applications to our system as well as to port our /net framework to more systems.

References

[1] W. Adjie-Winoto, E. Schwartz, H. Balakrishnan, and J. Lilley. The design and implementation of an intentional naming system. *Symposium on Operating Systems Principles*, 1999.

[2] S. Ahuja, N. Carriero, and D. Gelernter. Linda and friends. *IEEE Computer*, 19(8), Aug. 1986.

[3] D. Andersen, H. Balakrishnan, F. Kaashoek, and R. Morris. Resilient overlay networks. *Proceedings of the 18th SOSP*, 2001.

[4] F. J. Ballesteros, G. G. Muzquiz, K. L. Algara, E. Soriano, P. de las Heras Quirs, E. M. Castro, A. Leonardo, and S. Arvalo. Plan B: Boxes for network resources. *Journal of Brasilian Computer Society*, Special issue on Adaptable Computing Systems To appear.

[5] R. Grimm and B. Bershad. Future directions: System support for pervasive applications. *Proceedings of FuDiCo*, 2002.

[6] H. J. S. I. Kuz, M. Steen. The globe infrastructure directory service. *Computer Communications*, 25, 2002.

[7] LSUB. Plan b web site, 2001.

[8] A. S. T. M. Steen, P. Homburg. Globe: A wide-area distributed system. *IEEE Concurrency*, 1999.

[9] D. Maltz and P. Bhagwat. The jini architecture for network-centric computing. *INFOCOM*, (3), 1998.

[10] S. Microsystems. Java 2 platform standard edition, api specifi cation.

[11] 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 Principles*, 1997.

[12] T. Okoshi, M. Mochizuki, Y. Tobe, and H. Tokuda. Mobilesockets: Toward continuous operation for java applications. *Intl. Conference on Computer Communications and Networks*, 1999.

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

[14] D. Presotto and P. Winterbottom. The organization of networks in Plan 9. In *USENIX Association. Proceedings of the Winter 1993 USENIX Conference*, pages 271–280 (of x + 530), Berkeley, CA, USA, 1993. USENIX.

[15] S. Ratnasamy, P. Francis, M. Handley, R. Karp, and S. Shenker. A scalable content-addressable network. *Proc. ACM SIGCOMM*, 2001.

[16] J. H. Saltzer. On the naming and binding of network destinations. *Local Computer Networks. North-Holland Publishing Company. IFIP*, 1982.

[17] M. Stemm, R. H. Katz, and R. Morris. Vertical handoffs in wireless overlay networks. *Journal of Mobile Networks and Applications*, 3(4), 1998.

[18] Stevens. Unix network programming. *Prentice-Hall*, 1998.

[19] J. Walso. The jini architecture for network-centric computing. *Communications of the ACM*, 42(7), 1999.

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

[21] V. Zanty and B. Miller. Reliable network connections. *Mobicom 2002*, 2002.

[22] W.K. Edwards, M.W. Newman, J.Z. Sedivy, T.F. Smith, and S. Izadi. Challenge: Recombinant Computing and the Speakeasy Approach. *Proceedings of the 8th ACM Mobicom*, 2002.