Singularity Design Note 11

Name Server

This document contains the channel contract specification for the name server and relevant parts of the named services that wish to be registered and looked up through the name server. It also contains simplified sample implementations of these contracts.

1. Specification

The name server allows services to register and unregister themselves in a hierarchical namespace so that they can be discovered by clients. Its contract is given below.

```
contract NameService
{
    // input messages
    in message Register(List<String> path, imp<NameService> provider);
    in message Deregister(List<String> path);

    in message Lookup<C>(List<String> path, exp<C> ep);

    // output messages
    out message AckRegister;
    out message AckDeregister;
    out message AckLookup;

    out message NackRegister(imp<NameService> provider);
    out message NackDeregister;
    out message NackLookup<C>(exp<C> ep);

    // operations
    START: Request -> START;

    Request: one {
        Register? -> (AckRegister! or NackRegister!);
        Deregister? -> (AckDeregister! or Nack!);

        Lookup<C>? -> (AckLookup! or Nack!);
    }
};
```


The name space is hierarchical. Client programs access a service by passing a pathname to the server, obtaining a fresh channel for it. Conceptually, the name space consists of directories and services. Directories are used to organize the name space. A service, in the name server, is simply a means to get an endpoint to the actual service. Some example pathnames are "/filesystems/ntfs", "/tcp/128.0.0.1/80", and "/files/C/Work/proposal.doc".

The entire name space is not stored in one name server. Another name server may register itself to handle all requests below a particular point in the hierarchy. Register, deregister, and lookup messages for that subtree are forwarded to the helper name server. This functionality is similar to mount points in Unix file systems. However, the additional name servers need not function in the same way as a typical one; the TCP service above could expose a huge dynamic space of names without storing them in memory. Helper name servers could also be used to implement features like symbolic links.

1.1. Purpose of the Name Server

The contract given above is somewhat conservative in the features it offers. As it stands, the name server is capable only of mapping hierarchical names to endpoints. Since the mapping between names and channel endpoints is, in some ways, a form of dynamic metadata, it may be natural to extend the name server into a generic metadata store.

An extension like this one does not seem to present any fundamental challenges to specification of the name server by channel contracts. There are a few interesting issues related to typing, but they do not appear to be insurmountable. The rest of the document will relate only to the simple name server contract above.

1.2. Protocol Design

Clients interact with the name server by sending lookup requests to it. A fresh channel endpoint is included in the lookup request. If the lookup succeeds, then the client and server can communicate on the channel.

Services interact with the name server by responding to lookup requests. When a server registers with the name server, it supplies a fresh channel for lookup requests. Whenever a lookup request comes in, the service would most likely create a thread to handle client requests on the channel passed in with the lookup message.

The following chronology illustrates how the name server could be used. We'll assume that the processes **C**, **S**, and **NS** represent a client, a service, and a name service. We'll also assume that *nsC* and *nsS* are channels to the name server that the client and service have.

1. (**S** to **NS** on *nsS*) Register with fresh channel *lookup*
2. (**NS** to **S** on *nsS*) Registration acknowledgement
3. (**C** to **NS** on *nsC*) Lookup with fresh channel *service*
4. (**NS** to **S** on *lookup*) Lookup with *service*
5. (**S** to **NS** on *lookup*) Lookup reply
6. (**NS** to **C** on *nsC*) Lookup reply
7. **C** and **S** communicate using channel *service*

It's possible for services to expose more functionality to the name server. In the "/tcp/128.0.0.1/80" example, it would be ridiculous to have every IP address and port number combination registered with the name server. It would be better if the name server could contact a generic TCP service and pass the lookup request to it. The TCP service would create a thread to handle a TCP connection to a given host and would return an endpoint with which the client could communicate with this thread. Here's an example of how that would work.

1. (**S** to **NS** on *nsS*) Register("/tcp") with fresh channel *lookup*
2. (**NS** to **S** on *nsS*) Registration acknowledgement
3. (**C** to **NS** on *nsC*) Lookup("/tcp/128.0.0.1/80") with fresh channel *service*
4. (**NS** to **S** on *lookup*) Lookup("/128.0.0.1/80") with channel *service*

5. (**S** to **NS** on *lookup*) Lookup reply
6. (**NS** to **C** on *nsC*) Lookup reply
7. **C** and **S** send messages over channel *service*, and these messages are forwarded over TCP

1.3. Security Considerations

As the Singularity security model is unfinished, this section contains several proposals for name server security. The goal for any name server security model is to prevent a user from getting "bad" endpoints in lookup requests. Bad endpoints could be used to monitor user operations, steal passwords, and convince other processes to perform dangerous operations. Additionally, an insecure name server could be subject to denial of service attacks where authorized users are unable to lookup or register names. Finally, it may be desirable to forbid some users from lookup up certain names.

The simplest solution is to control who can register names and where they can register them. Placing access controls on each directory in the name server hierarchy, much like file system access controls, would prevent unauthorized registration. Users performing name lookups would have the assurance that the endpoint being looked up was created by someone with access to the parent directory. However, there are several important problems with this model.

- Since a name service can register itself inside the hierarchy of another name server, the entire path from root to leaf nodes must be protected. Otherwise, a rogue name server might register itself at a higher point in the path and then allow arbitrary operations to be performed farther down.
- Access control lists are a fairly coarse-grained way to control access to something as important as the name server.

One possible solution is to provide an operation where a program can determine a principal for the endpoint at the other end of a channel. Then, even if an unauthorized name were placed in the name hierarchy, a client program would discover that it does not recognize the principal at the other end of the channel returned from a lookup operation. An issue with this arrangement is that it's fairly static. If a program wants to ensure that the result of a lookup is valid, it must have prior knowledge of what it expects to get back. Users would be unable to substitute new programs or services for old ones, since the clients for those services might not recognize the new principal.

A more flexible option might work as follows. The name sever would have a "base" hierarchy that would be relatively fixed. This hierarchy would contain names for services like the file system, device drivers, network connections, and perhaps a user interface service (to display windows and such). The entire base hierarchy would have very strong access checks (similar to Administrator privileges) to ensure that the user would change it only in the event of a system upgrade. No new services could be added or changed anywhere in this name space without strong credentials.

However, above this hierarchy would be a set of overlay name spaces. Overlay name spaces would be layered atop one another forming a tree. Each overlay name space would keep track of the principal that created the overlay. Only that principal would be able to make changes to the overlay. When performing a lookup operation, a program would provide a list of trusted principals. In a simple system, this list might include the program itself, the user executing the program, as well as the principal for the base name space (which everyone would be forced to trust). The lookup operation would use only trusted overlays. An overlay would be trusted if its principal is trusted and if the overlay upon which it is layered is trusted. Security conscious programs could choose to trust only the base overlay and no others.

2. Client Implementation

It's not difficult for a client program to use a name server. First, it must obtain a channel to the name server. Then it performs a Lookup operation, passing it a pathname to the service and the channel over which it intends to communicate with the server. Assuming that the client already has a channel to the name server (it's conceivable that one would be provided at startup), the lookup code is as follows.

```
imp<NameService> ns = ...;
```



```
(imp<FSContract> ep, exp<FSContract> server) = new chan(FSContract);
send ns.Lookup(Path.pathOfString("/filesystems/ntfs"), server);
select {
    receive ns.AckLookup {
        // use ep according to FileSystemService contract
    }
    receive ns.NackLookup(exp<FSContract> server) {
        // abort the program or something
    }
}
```

3. Server Implementation

The mechanism through which servers interact with the name server is slightly more complex. They must register themselves and then respond to Lookup requests. Eventually, they may deregister. When registering, a service exports a NameService endpoint. The name server will use this endpoint to service requests. For simple services, only the Lookup message is pertinent and the others can be ignored. The registration process for an NTFS file system is shown.

```
imp<NameService> ns;
(imp<NameService> lookupImp, exp<NameService> lookupExp) = new chan(NameService);

send ns.Register(Path.pathOfString("/filesystems/ntfs"), lookupImp);
select {
    receive ns.AckRegister {
        AnswerLookups(lookupExp); // this function is supplied by the service
    }

    receive ns.NackRegister(imp<NameService> lookupImp) {
        lookupImp.close();
        System.abort("couldn't connect to name server");
    }
}
```

In this code, the ns endpoint is used to send registration requests to the name server. The lookup channel allows the name server to contact the service whenever a client looks up the service and needs a fresh channel to it. The service must answer requests on this channel.

AnswerLookups is a function that must respond to Lookup requests by creating new server processes.

```
void AnswerLookups(exp<NameService> lookup)
{
    while (running) {
        select {
            receive lookup.Lookup(List<String> path, exp<C> ep) {
                if (FileSystemContract conformsto C && path.length() == 0) {
                    StartServerThread(ep); // serve requests
                    send lookup.AckLookup;
                } else {
                    send lookup.NackLookup(ep);
                }
            }
        }
    }
}
```


A more sophisticated service would omit the "path.length() == 0" check and would expose an entire name space to the client. The path might be passed to StartServerThread() for its use.

