CptS 464/564
Lecture 18

2nd November 2004

Checkpoint

What have we covered so far? Paradigms and Models: frameworks for the discussion of DS

What is the plan ahead? Next: examples of distributed systems and platforms. Then real-time DS, security and management.

Chapter 4: Distributed Systems and Platforms: Examples

We turn now to looking at several examples of distributed systems, emphasizing how DS techniques are used to build platforms (common infrastructure) that other DS can be built upon. I will not lecture on section 4.4 concerning CORBA as we have already covered some of that material in more depth, and will come back to other aspects in another lecture.

Today we will look at naming and directory services. Next time, begin distributed file systems, the web, and other examples.

Naming Services

Recall: a naming service implements naming contexts that map names to values. The values themselves may be addresses, lower-level names, other naming contexts, etc., the particular collection of value types depending on the particular naming service.

A naming service may be implemented as a central server. Examples? Drawbacks? but most are implemented as a distributed system consisting of cooperating name servers. Why? Scalability. Management. Management is a particularly important issue: suppose each time we wanted to add or move a machine on the EECS network we had to interact with the Global Name Service Management Corporation (and so did the other 6B people in the world).

[Note argument about the *actual* scale on which this becomes a problem: 80,000 person corporation and 300,000 person corporation (attempt to) centrally manage their networks. They began doing so after concluding that distributed management was too costly in both in people time and in lack of consistent procedures and processes – and ultimately results. But clearly at the scale of the Internet central managment does not work.]

Indirection

DNS

The Internet Domain Name Service (DNS) is a well-known example of a naming service. As noted above, the main issue is scalability. To solve the scalability problem DNS *partitions* the name space hierarchically and *replicates* each portion of the name space. How much and where to replicate is a local management decision.

The other key to scalability is *caching* which is facilitated by the slowly-changing nature of most DNS data.

Issues in DNS: *authentication* and *authorization* of updates: if a DNS record is improperly updated clients may be directed to the wrong resources.

- Authentication: determining *who* is making a request
- Authorization: once you know who is making a request, determing whether or not they have permission to make the request

The Internet evolved from a relatively small community of cooperative organizations where these problems were less worrisome. Early versions of DNS protocol and early DNS implementations did not address them adequately. Continued existence of old code and old ways of operating even new code leave vulnerabilities.

Components of DNS: servers, often running a program called BIND (Berekeley Internet Name Daemon), but other implementations exist. Client programs called *resolvers* that use the DNS protocol to talk to servers. (Remember how we saw early on that the clients lookup of a hierarchical name involved stepping through a sequence of naming contexts, which may involve contacting several servers. The resolver is often implemented in a library function that programs can call to do name lookups.)

Another security problem, which is general enough to add to our list of differences between local and remote invocation: in local invocation, data passed between caller and callee is usually formatted by the language system, and neither caller nor callee has to concern itself with whether the data is well-formatted. In distributed systems, all components *must be prepared for ill-formed data*, that is, receiving messages that do not conform to the protocol. **Failure to do so continues to be one of the biggest sources of security problems in the Internet.**

Interestingly, while the DNS protocol specifies the interaction of clients and servers in detail, and describes how clients are to access replicated data, does nott address

the question of how to *implement* replication. Replication is usually accomplished by periodically transferring an entire DNS *zone* from its *primary* server to the *secondary servers* for the zone. This is an example of a *primary site* update algorithm, closely related in purpose and behavior to our discussion of *primary partition* techniques.

Shortcomings in DNS:

Hard to make wholesale changes to the domain structure: DNS's model is of single records which may be updated, rather than having some sort of data model for the entire name space.

Inverse mappings are hard: for example, to implement IP-address to host-name lookups (reverse-lookups) requires an entirely separate name hierarchy called *in-addr.arpa*, which must be maintained consistently with the forward mapping (name to address). Likewise, given the canonical name of a host, it is not really possible to find all its aliases, nor find all domains that a mail server serves, etc.

We take up the first of the shortcomings next with GNS and return to the second when we talk about directory services.

Note Microsoft's Windows Active Directory starts with DNS. At least in its early versions it suffered the "reorganization is hard" shortcoming of DNS. Having considerably expanded the use of DNS, this was much more of a problem for AD than for DNS. When last I looked at it, plans were that AD would have tools to make reorganization easier by automating the copying and modification of individual records, but not to incorporate aggregated data into the data model, allowing operations directly on the aggregates.

Global Name Service (GNS)

GNS was a design for a naming service by researchers at Digital Equipment Corp.'s System Research Center. Its data model is more amenable to managing structural changes. It also deals directly with questions of replication (but we won't go into that aspect).

Key operational problem: stability of names – on human time scales, names become entrenched in data and operational practices. Any change to a naming hierarchy that invalidates large numbers of names is potentially highly disruptive to an organization's operations.

Such problems are often sovled by building in a *layer of indirection.* Let's see how it works in GNS.

GNS names (another name for naming contexts) are always *relative* to a *working directory* (which may be the root directory) and carry, as part of the name, the *globally unique identifier (GUID)* of the directory.

- GUIDs really have to be unique on a global scale for this to work

- how can one get a GUID? Very large numbers, randomly chosen, are a useful approximation.

Combining two hierarchies: created a new root node with a new GUID. Make existing roots be children of the new root. (So far you could do these steps in DNS, and you would break existing names). Solution: new root must contain entries mapping the GUIDs of the old roots to their correct position in the new hierarchy. Thus the combination of using a GUID as a directory identifier and including a mapping from GUIDs to new positions in the root provides the required layer of indirection.

Of course, similar techniques could be used at any interior point in the directory hierarchy to accomodate a more local change. This is similar to the use of symbolic links in file systems.

Problem: scalability? Maybe, but notice that such reorganizations are relatively rare.

If a need for adaptability of a naming system is recognized, indirection may be part of the initial system design, as in GNS. More often, the need is only recognized when interoperability is required between two legacy systems. The result is often a layering of a more adaptable layer providing indirection over top of the existing systems. But sometimes it is done right from the outset: e.g. designers of URLs at least tried. I commend use of a *layer of indirection* to you as a very powerful technique for solving adaptability problems in distributed systems.

Directory services

X.500, LDAP, Active Directory, NDS

Name servers provide lookups of attributes from names; and as we saw with the reverse IP addr lookup, questions in the opposite direction are also quite natural, but name services are not adept at providing answers.

A directory service is a generalized name service that is able to map attributes to names (actually, collections of names) as well as names to attributes. Sometimes this is called a “yellow pages service” in contrast with traditional name services that are called “white pages services”. Note confusion: when Sun originally created NIS they called it Yellow Pages even though it was a traditional name service!

Bootstrapping and Discovery services

If you need a name service to resolve names to addresses and you need an address in order to talk to a name service, how do you get started? Indeed, what should you use as your own address for conversing with the name server?

How can you find a “nearby” printer?

Both are examples of a need for a *discovery service,* a naming or directory service that operates within a defined local scope, providing answers based on physical or network-topology based context. In some sense, there is no notion of rooted hierarchy for these services.

The scope is often set by default as "machines that will be reached by a LAN broadcast packet", though some protocols allow for so called *expanding-ring* broadcasts: nodes reachable in 1 hop, 2 hops, ... This latter approach is not popular in the Internet. Why?

Example: Jini - a Java-based framework for spontaneous networking. Jini relies on features of Java, the JVM and Java's RMI, including downloading code. It provides service discovery, transactions, shared dataspaces and events. We look only at service discovery.

Fundamental to the service discovery protocol is the *lookup service.* Jini services (similar to CORBA objects) register themselves in one or more Jini lookup services. Lookup service allows matching based on attributes including formal types (signatures) of Java interfaces. E.g. a Java interface for printing would contain specific methods taking particular parameters. Jini lookup service could identify a service exporting that interface.

Still have the bootstrapping problem: how does client find a lookup service?

- Configuration parameter (like we're doing with the CORBA name service)
- Client multicasts (or broadcasts) a request asking a name service to identify itself. This is also the technique used by protocols such as DHCP and BOOTP for configuring IP hosts – provides info such as DNS server.
- Server multicasts (or broadcasts) its existence periodically. Similar to the technique of IP hosts listening for routing table broadcasts in order to learn default router.

Finally, one more question: Jini is intended for use in *ad hoc,* spontaneous networking situations, perhaps involving mobile hosts. How do lookup services avoid becoming polluted with services that register then disappear? Answer: *leases.* Leasing is a common technique partially (a)synchronous networks: in a purely asynchronous framework they would not work and in a synchronous network they wouldn't be necessary. Idea: when a service registers with the lookup service, the lookup service promises to hold the registration for a fixed amount of time. If the the registration is not *renewed* before then, the registration expires. The lookup service no longer will include the registered service in its replies. Leasing is important in avoiding unnecessary scaling problems.

Thought problem: construct a convincing argument that leases will not work in a pure asynchronous framework and are not necessary in a pure synchronous framework.

Up next: Distributed File Systems