46

A Dialogue on Distribution

Professor: *And thus we reach our final little piece in the world of operating systems: distributed systems. Since we can't cover much here, we'll sneak in a little intro here in the section on persistence, and focus mostly on distributed file systems. Hope that is OK!*

Student: *Sounds OK. But what is a distributed system exactly, oh glorious and all-knowing professor?*

Professor: *Well, I bet you know how this is going to go...*

Student: *There's a peach?*

Professor: *Exactly! But this time, it's far away from you, and may take some time to get the peach. And there are a lot of them! Even worse, sometimes a peach becomes rotten. But you want to make sure that when anybody bites into a peach, they will get a mouthful of deliciousness.*

Student: *This peach analogy is working less and less for me.*

Professor: *Come on! It's the last one, just go with it.*

Student: *Fine.*

Professor: *So anyhow, forget about the peaches. Building distributed systems is hard, because things fail all the time. Messages get lost, machines go down, disks corrupt data. It's like the whole world is working against you!*

Student: *But I use distributed systems all the time, right?*

Professor: *Yes! You do. And... ?*

Student: *Well, it seems like they mostly work. After all, when I send a search request to google, it usually comes back in a snap, with some great results! Same thing when I use facebook, or Amazon, and so forth.*

Professor: *Yes, it is amazing. And that's despite all of those failures taking place! Those companies build a huge amount of machinery into their systems so as to ensure that even though some machines have failed, the entire system stays up and running. They use a lot of techniques to do this: replication, retry, and various other tricks people have developed over time to detect and recover from failures.*

Student: *Sounds interesting. Time to learn something for real?*

Professor: *It does seem so. Let's get to work! But first things first ...*
(bites into peach he has been holding, which unfortunately is rotten)

47

Distributed Systems

Distributed systems have changed the face of the world. When your web browser connects to a web server somewhere else on the planet, it is participating in what seems to be a simple form of a **client/server** distributed system. When you contact a modern web service such as Google or facebook, you are not just interacting with a single machine, however; behind the scenes, these complex services are built from a large collection (i.e., thousands) of machines, each of which cooperate to provide the particular service of the site. Thus, it should be clear what makes studying distributed systems interesting. Indeed, it is worthy of an entire class; here, we just introduce a few of the major topics.

A number of new challenges arise when building a distributed system. The major one we focus on is **failure**; machines, disks, networks, and software all fail from time to time, as we do not (and likely, will never) know how to build "perfect" components and systems. However, when we build a modern web service, we'd like it to appear to clients as if it never fails; how can we accomplish this task?

THE CRUX:
HOW TO BUILD SYSTEMS THAT WORK WHEN COMPONENTS FAIL
How can we build a working system out of parts that don't work correctly all the time? The basic question should remind you of some of the topics we discussed in RAID storage arrays; however, the problems here tend to be more complex, as are the solutions.

Interestingly, while failure is a central challenge in constructing distributed systems, it also represents an opportunity. Yes, machines fail; but the mere fact that a machine fails does not imply the entire system must fail. By collecting together a set of machines, we can build a system that appears to rarely fail, despite the fact that its components fail regularly. This reality is the central beauty and value of distributed systems, and why they underly virtually every modern web service you use, including Google, Facebook, etc.

TIP: COMMUNICATION IS INHERENTLY UNRELIABLE
In virtually all circumstances, it is good to view communication as a fundamentally unreliable activity. Bit corruption, down or non-working links and machines, and lack of buffer space for incoming packets all lead to the same result: packets sometimes do not reach their destination. To build reliable services atop such unreliable networks, we must consider techniques that can cope with packet loss.

Other important issues exist as well. System **performance** is often critical; with a network connecting our distributed system together, system designers must often think carefully about how to accomplish their given tasks, trying to reduce the number of messages sent and further make communication as efficient (low latency, high bandwidth) as possible.

Finally, **security** is also a necessary consideration. When connecting to a remote site, having some assurance that the remote party is who they say they are becomes a central problem. Further, ensuring that third parties cannot monitor or alter an on-going communication between two others is also a challenge.

In this introduction, we'll cover the most basic new aspect that is new in a distributed system: **communication**. Namely, how should machines within a distributed system communicate with one another? We'll start with the most basic primitives available, messages, and build a few higher-level primitives on top of them. As we said above, failure will be a central focus: how should communication layers handle failures?

47.1 Communication Basics

The central tenet of modern networking is that communication is fundamentally unreliable. Whether in the wide-area Internet, or a local-area high-speed network such as Infiniband, packets are regularly lost, corrupted, or otherwise do not reach their destination.

There are a multitude of causes for packet loss or corruption. Sometimes, during transmission, some bits get flipped due to electrical or other similar problems. Sometimes, an element in the system, such as a network link or packet router or even the remote host, are somehow damaged or otherwise not working correctly; network cables do accidentally get severed, at least sometimes.

More fundamental however is packet loss due to lack of buffering within a network switch, router, or endpoint. Specifically, even if we could guarantee that all links worked correctly, and that all the components in the system (switches, routers, end hosts) were up and running as expected, loss is still possible, for the following reason. Imagine a packet arrives at a router; for the packet to be processed, it must be placed in memory somewhere within the router. If many such packets arrive at

```
// client code
int main(int argc, char *argv[]) {
    int sd = UDP_Open(20000);
    struct sockaddr_in addr, addr2;
    int rc = UDP_FillSockAddr(&addr, "machine.cs.wisc.edu", 10000);
    char message[BUFFER_SIZE];
    sprintf(message, "hello world");
    rc = UDP_Write(sd, &addr, message, BUFFER_SIZE);
    if (rc > 0) {
        int rc = UDP_Read(sd, &addr2, buffer, BUFFER_SIZE);
    }
    return 0;
}

// server code
int main(int argc, char *argv[]) {
    int sd = UDP_Open(10000);
    assert(sd > -1);
    while (1) {
        struct sockaddr_in s;
        char buffer[BUFFER_SIZE];
        int rc = UDP_Read(sd, &s, buffer, BUFFER_SIZE);
        if (rc > 0) {
            char reply[BUFFER_SIZE];
            sprintf(reply, "reply");
            rc = UDP_Write(sd, &s, reply, BUFFER_SIZE);
        }
    }
    return 0;
}
```

Figure 47.1: **Example UDP/IP Client/Server Code**

once, it is possible that the memory within the router cannot accommodate all of the packets. The only choice the router has at that point is to **drop** one or more of the packets. This same behavior occurs at end hosts as well; when you send a large number of messages to a single machine, the machine's resources can easily become overwhelmed, and thus packet loss again arises.

Thus, packet loss is fundamental in networking. The question thus becomes: how should we deal with it?

47.2 Unreliable Communication Layers

One simple way is this: we don't deal with it. Because some applications know how to deal with packet loss, it is sometimes useful to let them communicate with a basic unreliable messaging layer, an example of the **end-to-end argument** one often hears about (see the **Aside** at end of chapter). One excellent example of such an unreliable layer is found in the **UDP/IP** networking stack available today on virtually all modern systems. To use UDP, a process uses the **sockets** API in order to create a **communication endpoint**; processes on other machines (or on the same machine) send UDP **datagrams** to the original process (a datagram is a fixed-sized message up to some max size).

```
int UDP_Open(int port) {
    int sd;
    if ((sd = socket(AF_INET, SOCK_DGRAM, 0)) == -1) { return -1; }
    struct sockaddr_in myaddr;
    bzero(&myaddr, sizeof(myaddr));
    myaddr.sin_family      = AF_INET;
    myaddr.sin_port        = htons(port);
    myaddr.sin_addr.s_addr = INADDR_ANY;
    if (bind(sd, (struct sockaddr *) &myaddr, sizeof(myaddr)) == -1) {
        close(sd);
        return -1;
    }
    return sd;
}

int UDP_FillSockAddr(struct sockaddr_in *addr, char *hostName, int port) {
    bzero(addr, sizeof(struct sockaddr_in));
    addr->sin_family = AF_INET;          // host byte order
    addr->sin_port   = htons(port);      // short, network byte order
    struct in_addr *inAddr;
    struct hostent *hostEntry;
    if ((hostEntry = gethostbyname(hostName)) == NULL) { return -1; }
    inAddr = (struct in_addr *) hostEntry->h_addr;
    addr->sin_addr = *inAddr;
    return 0;
}

int UDP_Write(int sd, struct sockaddr_in *addr, char *buffer, int n) {
    int addrLen = sizeof(struct sockaddr_in);
    return sendto(sd, buffer, n, 0, (struct sockaddr *) addr, addrLen);
}

int UDP_Read(int sd, struct sockaddr_in *addr, char *buffer, int n) {
  int len = sizeof(struct sockaddr_in);
  return recvfrom(sd, buffer, n, 0, (struct sockaddr *) addr,
                  (socklen_t *) &len);
}
```

Figure 47.2: **A Simple UDP Library**

Figures 47.1 and 47.2 show a simple client and server built on top of UDP/IP. The client can send a message to the server, which then responds with a reply. With this small amount of code, you have all you need to begin building distributed systems!

UDP is a great example of an unreliable communication layer. If you use it, you will encounter situations where packets get lost (dropped) and thus do not reach their destination; the sender is never thus informed of the loss. However, that does not mean that UDP does not guard against any failures at all. For example, UDP includes a **checksum** to detect some forms of packet corruption.

However, because many applications simply want to send data to a destination and not worry about packet loss, we need more. Specifically, we need reliable communication on top of an unreliable network.

TIP: USE CHECKSUMS FOR INTEGRITY

Checksums are a commonly-used method to detect corruption quickly and effectively in modern systems. A simple checksum is addition: just sum up the bytes of a chunk of data; of course, many other more sophisticated checksums have been created, including basic cyclic redundancy codes (CRCs), the Fletcher checksum, and many others [MK09].

In networking, checksums are used as follows. Before sending a message from one machine to another, compute a checksum over the bytes of the message. Then send both the message and the checksum to the destination. At the destination, the receiver computes a checksum over the incoming message as well; if this computed checksum matches the sent checksum, the receiver can feel some assurance that the data likely did not get corrupted during transmission.

Checksums can be evaluated along a number of different axes. Effectiveness is one primary consideration: does a change in the data lead to a change in the checksum? The stronger the checksum, the harder it is for changes in the data to go unnoticed. Performance is the other important criterion: how costly is the checksum to compute? Unfortunately, effectiveness and performance are often at odds, meaning that checksums of high quality are often expensive to compute. Life, again, isn't perfect.

47.3 Reliable Communication Layers

To build a reliable communication layer, we need some new mechanisms and techniques to handle packet loss. Let us consider a simple example in which a client is sending a message to a server over an unreliable connection. The first question we must answer: how does the sender know that the receiver has actually received the message?

The technique that we will use is known as an **acknowledgment**, or **ack** for short. The idea is simple: the sender sends a message to the receiver; the receiver then sends a short message back to *acknowledge* its receipt. Figure 47.3 depicts the process.

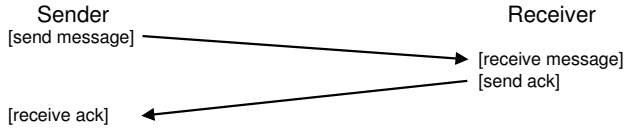

Figure 47.3: **Message Plus Acknowledgment**

When the sender receives an acknowledgment of the message, it can then rest assured that the receiver did indeed receive the original message. However, what should the sender do if it does not receive an acknowledgment?

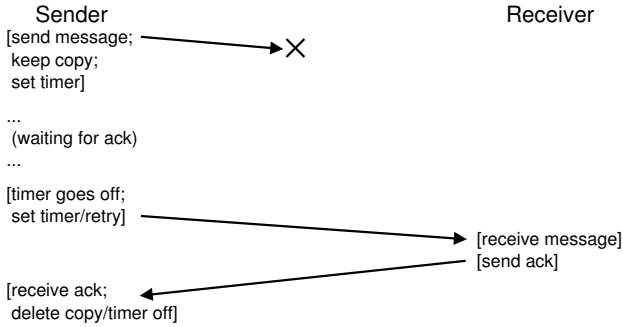

Figure 47.4: **Message Plus Acknowledgment: Dropped Request**

To handle this case, we need an additional mechanism, known as a **timeout**. When the sender sends a message, the sender now sets a timer to go off after some period of time. If, in that time, no acknowledgment has been received, the sender concludes that the message has been lost. The sender then simply performs a **retry** of the send, sending the same message again with hopes that this time, it will get through. For this approach to work, the sender must keep a copy of the message around, in case it needs to send it again. The combination of the timeout and the retry have led some to call the approach **timeout/retry**; pretty clever crowd, those networking types, no? Figure 47.4 shows an example.

Unfortunately, timeout/retry in this form is not quite enough. Figure 47.5 shows an example of packet loss which could lead to trouble. In this example, it is not the original message that gets lost, but the acknowledgment. From the perspective of the sender, the situation seems the same: no ack was received, and thus a timeout and retry are in order. But from the perspective of the receiver, it is quite different: now the same message has been received twice! While there may be cases where this is OK, in general it is not; imagine what would happen when you are downloading a file and extra packets are repeated inside the download. Thus, when we are aiming for a reliable message layer, we also usually want to guarantee that each message is received **exactly once** by the receiver.

To enable the receiver to detect duplicate message transmission, the sender has to identify each message in some unique way, and the receiver needs some way to track whether it has already seen each message before. When the receiver sees a duplicate transmission, it simply acks the message, but (critically) does *not* pass the message to the application that receives the data. Thus, the sender receives the ack but the message is not received twice, preserving the exactly-once semantics mentioned above.

There are myriad ways to detect duplicate messages. For example, the sender could generate a unique ID for each message; the receiver could track every ID it has ever seen. This approach could work, but it is prohibitively costly, requiring unbounded memory to track all IDs.

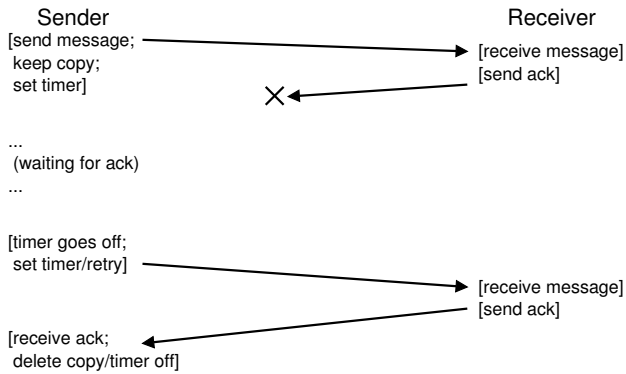

Figure 47.5: **Message Plus Acknowledgment: Dropped Reply**

A simpler approach, requiring little memory, solves this problem, and the mechanism is known as a **sequence counter**. With a sequence counter, the sender and receiver agree upon a start value (e.g., 1) for a counter that each side will maintain. Whenever a message is sent, the current value of the counter is sent along with the message; this counter value (N) serves as an ID for the message. After the message is sent, the sender then increments the value (to $N + 1$).

The receiver uses its counter value as the expected value for the ID of the incoming message from that sender. If the ID of a received message (N) matches the receiver's counter (also N), it acks the message and passes it up to the application; in this case, the receiver concludes this is the first time this message has been received. The receiver then increments its counter (to $N + 1$), and waits for the next message.

If the ack is lost, the sender will timeout and re-send message N. This time, the receiver's counter is higher ($N+1$), and thus the receiver knows it has already received this message. Thus it acks the message but does *not* pass it up to the application. In this simple manner, sequence counters can be used to avoid duplicates.

The most commonly used reliable communication layer is known as **TCP/IP**, or just **TCP** for short. TCP has a great deal more sophistication than we describe above, including machinery to handle congestion in the network [VJ88], multiple outstanding requests, and hundreds of other small tweaks and optimizations. Read more about it if you're curious; better yet, take a networking course and learn that material well.

47.4 Communication Abstractions

Given a basic messaging layer, we now approach the next question in this chapter: what abstraction of communication should we use when building a distributed system?

TIP: BE CAREFUL SETTING THE TIMEOUT VALUE

As you can probably guess from the discussion, setting the timeout value correctly is an important aspect of using timeouts to retry message sends. If the timeout is too small, the sender will re-send messages needlessly, thus wasting CPU time on the sender and network resources. If the timeout is too large, the sender waits too long to re-send and thus perceived performance at the sender is reduced. The "right" value, from the perspective of a single client and server, is thus to wait just long enough to detect packet loss but no longer.

However, there are often more than just a single client and server in a distributed system, as we will see in future chapters. In a scenario with many clients sending to a single server, packet loss at the server may be an indicator that the server is overloaded. If true, clients might retry in a different adaptive manner; for example, after the first timeout, a client might increase its timeout value to a higher amount, perhaps twice as high as the original value. Such an **exponential back-off** scheme, pioneered in the early Aloha network and adopted in early Ethernet [A70], avoids situations where resources are being overloaded by an excess of re-sends. Robust systems strive to avoid overload of this nature.

The systems community developed a number of approaches over the years. One body of work took OS abstractions and extended them to operate in a distributed environment. For example, **distributed shared memory** (**DSM**) systems enable processes on different machines to share a large, virtual address space [LH89]. This abstraction turns a distributed computation into something that looks like a multi-threaded application; the only difference is that these threads run on different machines instead of different processors within the same machine.

The way most DSM systems work is through the virtual memory system of the OS. When a page is accessed on one machine, two things can happen. In the first (best) case, the page is already local on the machine, and thus the data is fetched quickly. In the second case, the page is currently on some other machine. A page fault occurs, and the page fault handler sends a message to some other machine to fetch the page, install it in the page table of the requesting process, and continue execution.

This approach is not widely in use today for a number of reasons. The largest problem for DSM is how it handles failure. Imagine, for example, if a machine fails; what happens to the pages on that machine? What if the data structures of the distributed computation are spread across the entire address space? In this case, parts of these data structures would suddenly become unavailable. Dealing with failure when parts of your address space go missing is hard; imagine a linked list that where a next pointer points into a portion of the address space that is gone. Yikes!

A further problem is performance. One usually assumes, when writing code, that access to memory is cheap. In DSM systems, some accesses

are inexpensive, but others cause page faults and expensive fetches from remote machines. Thus, programmers of such DSM systems had to be very careful to organize computations such that almost no communication occurred at all, defeating much of the point of such an approach. Though much research was performed in this space, there was little practical impact; nobody builds reliable distributed systems using DSM today.

47.5 Remote Procedure Call (RPC)

While OS abstractions turned out to be a poor choice for building distributed systems, programming language (PL) abstractions make much more sense. The most dominant abstraction is based on the idea of a **remote procedure call**, or **RPC** for short [BN84][1].

Remote procedure call packages all have a simple goal: to make the process of executing code on a remote machine as simple and straightforward as calling a local function. Thus, to a client, a procedure call is made, and some time later, the results are returned. The server simply defines some routines that it wishes to export. The rest of the magic is handled by the RPC system, which in general has two pieces: a **stub generator** (sometimes called a **protocol compiler**), and the **run-time library**. We'll now take a look at each of these pieces in more detail.

Stub Generator

The stub generator's job is simple: to remove some of the pain of packing function arguments and results into messages by automating it. Numerous benefits arise: one avoids, by design, the simple mistakes that occur in writing such code by hand; further, a stub compiler can perhaps optimize such code and thus improve performance.

The input to such a compiler is simply the set of calls a server wishes to export to clients. Conceptually, it could be something as simple as this:

```
interface {
  int func1(int arg1);
  int func2(int arg1, int arg2);
};
```

The stub generator takes an interface like this and generates a few different pieces of code. For the client, a **client stub** is generated, which contains each of the functions specified in the interface; a client program wishing to use this RPC service would link with this client stub and call into it in order to make RPCs.

Internally, each of these functions in the client stub do all of the work needed to perform the remote procedure call. To the client, the code just

[1] In modern programming languages, we might instead say **remote method invocation** (**RMI**), but who likes these languages anyhow, with all of their fancy objects?

appears as a function call (e.g., the client calls `func1(x)`); internally, the code in the client stub for `func1()` does this:

- **Create a message buffer.** A message buffer is usually just a contiguous array of bytes of some size.
- **Pack the needed information into the message buffer.** This information includes some kind of identifier for the function to be called, as well as all of the arguments that the function needs (e.g., in our example above, one integer for `func1`). The process of putting all of this information into a single contiguous buffer is sometimes referred to as the **marshaling** of arguments or the **serialization** of the message.
- **Send the message to the destination RPC server.** The communication with the RPC server, and all of the details required to make it operate correctly, are handled by the RPC run-time library, described further below.
- **Wait for the reply.** Because function calls are usually **synchronous**, the call will wait for its completion.
- **Unpack return code and other arguments.** If the function just returns a single return code, this process is straightforward; however, more complex functions might return more complex results (e.g., a list), and thus the stub might need to unpack those as well. This step is also known as **unmarshaling** or **deserialization**.
- **Return to the caller.** Finally, just return from the client stub back into the client code.

For the server, code is also generated. The steps taken on the server are as follows:

- **Unpack the message.** This step, called **unmarshaling** or **deserialization**, takes the information out of the incoming message. The function identifier and arguments are extracted.
- **Call into the actual function.** Finally! We have reached the point where the remote function is actually executed. The RPC runtime calls into the function specified by the ID and passes in the desired arguments.
- **Package the results.** The return argument(s) are marshaled back into a single reply buffer.
- **Send the reply.** The reply is finally sent to the caller.

There are a few other important issues to consider in a stub compiler. The first is complex arguments, i.e., how does one package and send a complex data structure? For example, when one calls the `write()` system call, one passes in three arguments: an integer file descriptor, a pointer to a buffer, and a size indicating how many bytes (starting at the pointer) are to be written. If an RPC package is passed a pointer, it needs to be able to figure out how to interpret that pointer, and perform the

correct action. Usually this is accomplished through either well-known types (e.g., a `buffer_t` that is used to pass chunks of data given a size, which the RPC compiler understands), or by annotating the data structures with more information, enabling the compiler to know which bytes need to be serialized.

Another important issue is the organization of the server with regards to concurrency. A simple server just waits for requests in a simple loop, and handles each request one at a time. However, as you might have guessed, this can be grossly inefficient; if one RPC call blocks (e.g., on I/O), server resources are wasted. Thus, most servers are constructed in some sort of concurrent fashion. A common organization is a **thread pool**. In this organization, a finite set of threads are created when the server starts; when a message arrives, it is dispatched to one of these worker threads, which then does the work of the RPC call, eventually replying; during this time, a main thread keeps receiving other requests, and perhaps dispatching them to other workers. Such an organization enables concurrent execution within the server, thus increasing its utilization; the standard costs arise as well, mostly in programming complexity, as the RPC calls may now need to use locks and other synchronization primitives in order to ensure their correct operation.

Run-Time Library

The run-time library handles much of the heavy lifting in an RPC system; most performance and reliability issues are handled herein. We'll now discuss some of the major challenges in building such a run-time layer.

One of the first challenges we must overcome is how to locate a remote service. This problem, of **naming**, is a common one in distributed systems, and in some sense goes beyond the scope of our current discussion. The simplest of approaches build on existing naming systems, e.g., hostnames and port numbers provided by current internet protocols. In such a system, the client must know the hostname or IP address of the machine running the desired RPC service, as well as the port number it is using (a port number is just a way of identifying a particular communication activity taking place on a machine, allowing multiple communication channels at once). The protocol suite must then provide a mechanism to route packets to a particular address from any other machine in the system. For a good discussion of naming, you'll have to look elsewhere, e.g., read about DNS and name resolution on the Internet, or better yet just read the excellent chapter in Saltzer and Kaashoek's book [SK09].

Once a client knows which server it should talk to for a particular remote service, the next question is which transport-level protocol should RPC be built upon. Specifically, should the RPC system use a reliable protocol such as TCP/IP, or be built upon an unreliable communication layer such as UDP/IP?

Naively the choice would seem easy: clearly we would like for a request to be reliably delivered to the remote server, and clearly we would

like to reliably receive a reply. Thus we should choose the reliable transport protocol such as TCP, right?

Unfortunately, building RPC on top of a reliable communication layer can lead to a major inefficiency in performance. Recall from the discussion above how reliable communication layers work: with acknowledgments plus timeout/retry. Thus, when the client sends an RPC request to the server, the server responds with an acknowledgment so that the caller knows the request was received. Similarly, when the server sends the reply to the client, the client acks it so that the server knows it was received. By building a request/response protocol (such as RPC) on top of a reliable communication layer, two "extra" messages are sent.

For this reason, many RPC packages are built on top of unreliable communication layers, such as UDP. Doing so enables a more efficient RPC layer, but does add the responsibility of providing reliability to the RPC system. The RPC layer achieves the desired level of responsibility by using timeout/retry and acknowledgments much like we described above. By using some form of sequence numbering, the communication layer can guarantee that each RPC takes place exactly once (in the case of no failure), or at most once (in the case where failure arises).

Other Issues

There are some other issues an RPC run-time must handle as well. For example, what happens when a remote call takes a long time to complete? Given our timeout machinery, a long-running remote call might appear as a failure to a client, thus triggering a retry, and thus the need for some care here. One solution is to use an explicit acknowledgment (from the receiver to sender) when the reply isn't immediately generated; this lets the client know the server received the request. Then, after some time has passed, the client can periodically ask whether the server is still working on the request; if the server keeps saying "yes", the client should be happy and continue to wait (after all, sometimes a procedure call can take a long time to finish executing).

The run-time must also handle procedure calls with large arguments, larger than what can fit into a single packet. Some lower-level network protocols provide such sender-side **fragmentation** (of larger packets into a set of smaller ones) and receiver-side **reassembly** (of smaller parts into one larger logical whole); if not, the RPC run-time may have to implement such functionality itself. See Birrell and Nelson's excellent RPC paper for details [BN84].

One issue that many systems handle is that of **byte ordering**. As you may know, some machines store values in what is known as **big endian** ordering, whereas others use **little endian** ordering. Big endian stores bytes (say, of an integer) from most significant to least significant bits, much like Arabic numerals; little endian does the opposite. Both are equally valid ways of storing numeric information; the question here is how to communicate between machines of *different* endianness.

Aside: **The End-to-End Argument**

The **end-to-end argument** makes the case that the highest level in a system, i.e., usually the application at "the end", is ultimately the only locale within a layered system where certain functionality can truly be implemented. In their landmark paper, Saltzer et al. argue this through an excellent example: reliable file transfer between two machines. If you want to transfer a file from machine A to machine B, and make sure that the bytes that end up on B are exactly the same as those that began on A, you must have an "end-to-end" check of this; lower-level reliable machinery, e.g., in the network or disk, provides no such guarantee.

The contrast is an approach which tries to solve the reliable-file-transfer problem by adding reliability to lower layers of the system. For example, say we build a reliable communication protocol and use it to build our reliable file transfer. The communication protocol guarantees that every byte sent by a sender will be received in order by the receiver, say using timeout/retry, acknowledgments, and sequence numbers. Unfortunately, using such a protocol does not a reliable file transfer make; imagine the bytes getting corrupted in sender memory before the communication even takes place, or something bad happening when the receiver writes the data to disk. In those cases, even though the bytes were delivered reliably across the network, our file transfer was ultimately not reliable. To build a reliable file transfer, one must include end-to-end checks of reliability, e.g., after the entire transfer is complete, read back the file on the receiver disk, compute a checksum, and compare that checksum to that of the file on the sender.

The corollary to this maxim is that sometimes having lower layers provide extra functionality can indeed improve system performance or otherwise optimize a system. Thus, you should not rule out having such machinery at a lower-level in a system; rather, you should carefully consider the utility of such machinery, given its eventual usage in an overall system or application.

RPC packages often handle this by providing a well-defined endianness within their message formats. In Sun's RPC package, the **XDR** (**eXternal Data Representation**) layer provides this functionality. If the machine sending or receiving a message matches the endianness of XDR, messages are just sent and received as expected. If, however, the machine communicating has a different endianness, each piece of information in the message must be converted. Thus, the difference in endianness can have a small performance cost.

A final issue is whether to expose the asynchronous nature of communication to clients, thus enabling some performance optimizations. Specifically, typical RPCs are made **synchronously**, i.e., when a client issues the procedure call, it must wait for the procedure call to return before continuing. Because this wait can be long, and because the client

may have other work it could be doing, some RPC packages enable you to invoke an RPC **asynchronously**. When an asynchronous RPC is issued, the RPC package sends the request and returns immediately; the client is then free to do other work, such as call other RPCs or other useful computation. The client at some point will want to see the results of the asynchronous RPC; it thus calls back into the RPC layer, telling it to wait for outstanding RPCs to complete, at which point return arguments can be accessed.

47.6 Summary

We have seen the introduction of a new topic, distributed systems, and its major issue: how to handle failure which is now a commonplace event. As they say inside of Google, when you have just your desktop machine, failure is rare; when you're in a data center with thousands of machines, failure is happening all the time. The key to any distributed system is how you deal with that failure.

We have also seen that communication forms the heart of any distributed system. A common abstraction of that communication is found in remote procedure call (RPC), which enables clients to make remote calls on servers; the RPC package handles all of the gory details, including timeout/retry and acknowledgment, in order to deliver a service that closely mirrors a local procedure call.

The best way to really understand an RPC package is of course to use one yourself. Sun's RPC system, using the stub compiler `rpcgen`, is a common one, and is widely available on systems today, including Linux. Try it out, and see what all the fuss is about.

References

[A70] "The ALOHA System — Another Alternative for Computer Communications"
Norman Abramson
The 1970 Fall Joint Computer Conference
The ALOHA network pioneered some basic concepts in networking, including exponential back-off and retransmit, which formed the basis for communication in shared-bus Ethernet networks for years.

[BN84] "Implementing Remote Procedure Calls"
Andrew D. Birrell, Bruce Jay Nelson
ACM TOCS, Volume 2:1, February 1984
The foundational RPC system upon which all others build. Yes, another pioneering effort from our friends at Xerox PARC.

[MK09] "The Effectiveness of Checksums for Embedded Control Networks"
Theresa C. Maxino and Philip J. Koopman
IEEE Transactions on Dependable and Secure Computing, 6:1, January '09
A nice overview of basic checksum machinery and some performance and robustness comparisons between them.

[LH89] "Memory Coherence in Shared Virtual Memory Systems"
Kai Li and Paul Hudak
ACM TOCS, 7:4, November 1989
The introduction of software-based shared memory via virtual memory. An intriguing idea for sure, but not a lasting or good one in the end.

[SK09] "Principles of Computer System Design"
Jerome H. Saltzer and M. Frans Kaashoek
Morgan-Kaufmann, 2009
An excellent book on systems, and a must for every bookshelf. One of the few terrific discussions on naming we've seen.

[SRC84] "End-To-End Arguments in System Design"
Jerome H. Saltzer, David P. Reed, David D. Clark
ACM TOCS, 2:4, November 1984
A beautiful discussion of layering, abstraction, and where functionality must ultimately reside in computer systems.

[VJ88] "Congestion Avoidance and Control"
Van Jacobson
SIGCOMM '88
A pioneering paper on how clients should adjust to perceived network congestion; definitely one of the key pieces of technology underlying the Internet, and a must read for anyone serious about systems, and for Van Jacobson's relatives because well relatives should read all of your papers.

48

Sun's Network File System (NFS)

One of the first uses of distributed client/server computing was in the realm of distributed file systems. In such an environment, there are a number of client machines and one server (or a few); the server stores the data on its disks, and clients request data through well-formed protocol messages. Figure 48.1 depicts the basic setup.

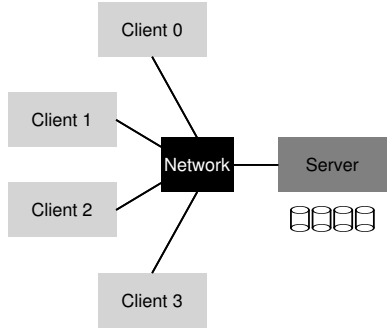

Figure 48.1: **A Generic Client/Server System**

As you can see from the picture, the server has the disks, and clients send messages across a network to access their directories and files on those disks. Why do we bother with this arrangement? (i.e., why don't we just let clients use their local disks?) Well, primarily this setup allows for easy **sharing** of data across clients. Thus, if you access a file on one machine (Client 0) and then later use another (Client 2), you will have the same view of the file system. Your data is naturally shared across these different machines. A secondary benefit is **centralized administration**; for example, backing up files can be done from the few server machines instead of from the multitude of clients. Another advantage could be **security**; having all servers in a locked machine room prevents certain types of problems from arising.

CRUX: HOW TO BUILD A DISTRIBUTED FILE SYSTEM
How do you build a distributed file system? What are the key aspects to think about? What is easy to get wrong? What can we learn from existing systems?

48.1 A Basic Distributed File System

We now will study the architecture of a simplified distributed file system. A simple client/server distributed file system has more components than the file systems we have studied so far. On the client side, there are client applications which access files and directories through the **client-side file system**. A client application issues **system calls** to the client-side file system (such as `open()`, `read()`, `write()`, `close()`, `mkdir()`, etc.) in order to access files which are stored on the server. Thus, to client applications, the file system does not appear to be any different than a local (disk-based) file system, except perhaps for performance; in this way, distributed file systems provide **transparent** access to files, an obvious goal; after all, who would want to use a file system that required a different set of APIs or otherwise was a pain to use?

The role of the client-side file system is to execute the actions needed to service those system calls. For example, if the client issues a `read()` request, the client-side file system may send a message to the **server-side file system** (or, as it is commonly called, the **file server**) to read a particular block; the file server will then read the block from disk (or its own in-memory cache), and send a message back to the client with the requested data. The client-side file system will then copy the data into the user buffer supplied to the `read()` system call and thus the request will complete. Note that a subsequent `read()` of the same block on the client may be **cached** in client memory or on the client's disk even; in the best such case, no network traffic need be generated.

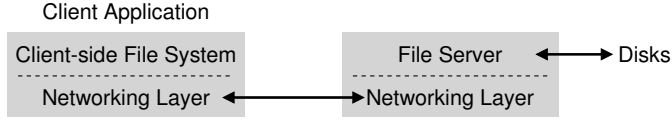

Figure 48.2: **Distributed File System Architecture**

From this simple overview, you should get a sense that there are two important pieces of software in a client/server distributed file system: the client-side file system and the file server. Together their behavior determines the behavior of the distributed file system. Now it's time to study one particular system: Sun's Network File System (NFS).

ASIDE: **WHY SERVERS CRASH**

Before getting into the details of the NFSv2 protocol, you might be wondering: why do servers crash? Well, as you might guess, there are plenty of reasons. Servers may simply suffer from a **power outage** (temporarily); only when power is restored can the machines be restarted. Servers are often comprised of hundreds of thousands or even millions of lines of code; thus, they have **bugs** (even good software has a few bugs per hundred or thousand lines of code), and thus they eventually will trigger a bug that will cause them to crash. They also have memory leaks; even a small memory leak will cause a system to run out of memory and crash. And, finally, in distributed systems, there is a network between the client and the server; if the network acts strangely (for example, if it becomes **partitioned** and clients and servers are working but cannot communicate), it may appear as if a remote machine has crashed, but in reality it is just not currently reachable through the network.

48.2 On To NFS

One of the earliest and quite successful distributed systems was developed by Sun Microsystems, and is known as the Sun Network File System (or NFS) [S86]. In defining NFS, Sun took an unusual approach: instead of building a proprietary and closed system, Sun instead developed an **open protocol** which simply specified the exact message formats that clients and servers would use to communicate. Different groups could develop their own NFS servers and thus compete in an NFS marketplace while preserving interoperability. It worked: today there are many companies that sell NFS servers (including Oracle/Sun, NetApp [HLM94], EMC, IBM, and others), and the widespread success of NFS is likely attributed to this "open market" approach.

48.3 Focus: Simple and Fast Server Crash Recovery

In this chapter, we will discuss the classic NFS protocol (version 2, a.k.a. NFSv2), which was the standard for many years; small changes were made in moving to NFSv3, and larger-scale protocol changes were made in moving to NFSv4. However, NFSv2 is both wonderful and frustrating and thus serves as our focus.

In NFSv2, the main goal in the design of the protocol was *simple and fast server crash recovery*. In a multiple-client, single-server environment, this goal makes a great deal of sense; any minute that the server is down (or unavailable) makes *all* the client machines (and their users) unhappy and unproductive. Thus, as the server goes, so goes the entire system.

48.4 Key To Fast Crash Recovery: Statelessness

This simple goal is realized in NFSv2 by designing what we refer to as a **stateless** protocol. The server, by design, does not keep track of anything about what is happening at each client. For example, the server does not know which clients are caching which blocks, or which files are currently open at each client, or the current file pointer position for a file, etc. Simply put, the server does not track anything about what clients are doing; rather, the protocol is designed to deliver in each protocol request *all the information* that is needed in order to complete the request. If it doesn't now, this stateless approach will make more sense as we discuss the protocol in more detail below.

For an example of a **stateful** (not stateless) protocol, consider the `open()` system call. Given a pathname, `open()` returns a file descriptor (an integer). This descriptor is used on subsequent `read()` or `write()` requests to access various file blocks, as in this application code (note that proper error checking of the system calls is omitted for space reasons):

```
char buffer[MAX];
int fd = open("foo", O_RDONLY); // get descriptor "fd"
read(fd, buffer, MAX);          // read MAX bytes from foo (via fd)
read(fd, buffer, MAX);          // read MAX bytes from foo
...
read(fd, buffer, MAX);          // read MAX bytes from foo
close(fd);                      // close file
```

Figure 48.3: **Client Code: Reading From A File**

Now imagine that the client-side file system opens the file by sending a protocol message to the server saying "open the file 'foo' and give me back a descriptor". The file server then opens the file locally on its side and sends the descriptor back to the client. On subsequent reads, the client application uses that descriptor to call the `read()` system call; the client-side file system then passes the descriptor in a message to the file server, saying "read some bytes from the file that is referred to by the descriptor I am passing you here".

In this example, the file descriptor is a piece of **shared state** between the client and the server (Ousterhout calls this **distributed state** [O91]). Shared state, as we hinted above, complicates crash recovery. Imagine the server crashes after the first read completes, but before the client has issued the second one. After the server is up and running again, the client then issues the second read. Unfortunately, the server has no idea to which file `fd` is referring; that information was ephemeral (i.e., in memory) and thus lost when the server crashed. To handle this situation, the client and server would have to engage in some kind of **recovery protocol**, where the client would make sure to keep enough information around in its memory to be able to tell the server what it needs to know (in this case, that file descriptor `fd` refers to file `foo`).

It gets even worse when you consider the fact that a stateful server has to deal with client crashes. Imagine, for example, a client that opens a file and then crashes. The `open()` uses up a file descriptor on the server; how can the server know it is OK to close a given file? In normal operation, a client would eventually call `close()` and thus inform the server that the file should be closed. However, when a client crashes, the server never receives a `close()`, and thus has to notice the client has crashed in order to close the file.

For these reasons, the designers of NFS decided to pursue a stateless approach: each client operation contains all the information needed to complete the request. No fancy crash recovery is needed; the server just starts running again, and a client, at worst, might have to retry a request.

48.5 The NFSv2 Protocol

We thus arrive at the NFSv2 protocol definition. Our problem statement is simple:

> THE CRUX: HOW TO DEFINE A STATELESS FILE PROTOCOL
> How can we define the network protocol to enable stateless operation? Clearly, stateful calls like `open()` can't be a part of the discussion (as it would require the server to track open files); however, the client application will want to call `open()`, `read()`, `write()`, `close()` and other standard API calls to access files and directories. Thus, as a refined question, how do we define the protocol to both be stateless *and* support the POSIX file system API?

One key to understanding the design of the NFS protocol is understanding the **file handle**. File handles are used to uniquely describe the file or directory a particular operation is going to operate upon; thus, many of the protocol requests include a file handle.

You can think of a file handle as having three important components: a *volume identifier*, an *inode number*, and a *generation number*; together, these three items comprise a unique identifier for a file or directory that a client wishes to access. The volume identifier informs the server which file system the request refers to (an NFS server can export more than one file system); the inode number tells the server which file within that partition the request is accessing. Finally, the generation number is needed when reusing an inode number; by incrementing it whenever an inode number is reused, the server ensures that a client with an old file handle can't accidentally access the newly-allocated file.

Here is a summary of some of the important pieces of the protocol; the full protocol is available elsewhere (see Callaghan's book for an excellent and detailed overview of NFS [C00]).

```
NFSPROC_GETATTR
  expects: file handle
  returns: attributes
NFSPROC_SETATTR
  expects: file handle, attributes
  returns: nothing
NFSPROC_LOOKUP
  expects: directory file handle, name of file/directory to look up
  returns: file handle
NFSPROC_READ
  expects: file handle, offset, count
  returns: data, attributes
NFSPROC_WRITE
  expects: file handle, offset, count, data
  returns: attributes
NFSPROC_CREATE
  expects: directory file handle, name of file, attributes
  returns: nothing
NFSPROC_REMOVE
  expects: directory file handle, name of file to be removed
  returns: nothing
NFSPROC_MKDIR
  expects: directory file handle, name of directory, attributes
  returns: file handle
NFSPROC_RMDIR
  expects: directory file handle, name of directory to be removed
  returns: nothing
NFSPROC_READDIR
  expects: directory handle, count of bytes to read, cookie
  returns: directory entries, cookie (to get more entries)
```

Figure 48.4: **The NFS Protocol: Examples**

We briefly highlight the important components of the protocol. First, the LOOKUP protocol message is used to obtain a file handle, which is then subsequently used to access file data. The client passes a directory file handle and name of a file to look up, and the handle to that file (or directory) plus its attributes are passed back to the client from the server.

For example, assume the client already has a directory file handle for the root directory of a file system (`/`) (indeed, this would be obtained through the NFS **mount protocol**, which is how clients and servers first are connected together; we do not discuss the mount protocol here for sake of brevity). If an application running on the client opens the file `/foo.txt`, the client-side file system sends a lookup request to the server, passing it the root file handle and the name `foo.txt`; if successful, the file handle (and attributes) for `foo.txt` will be returned.

In case you are wondering, attributes are just the metadata that the file system tracks about each file, including fields such as file creation time, last modification time, size, ownership and permissions information, and so forth, i.e., the same type of information that you would get back if you called `stat()` on a file.

Once a file handle is available, the client can issue READ and WRITE protocol messages on a file to read or write the file, respectively. The READ protocol message requires the protocol to pass along the file handle

of the file along with the offset within the file and number of bytes to read. The server then will be able to issue the read (after all, the handle tells the server which volume and which inode to read from, and the offset and count tells it which bytes of the file to read) and return the data to the client (or an error if there was a failure). WRITE is handled similarly, except the data is passed from the client to the server, and just a success code is returned.

One last interesting protocol message is the GETATTR request; given a file handle, it simply fetches the attributes for that file, including the last modified time of the file. We will see why this protocol request is important in NFSv2 below when we discuss caching (can you guess why?).

48.6 From Protocol to Distributed File System

Hopefully you are now getting some sense of how this protocol is turned into a file system across the client-side file system and the file server. The client-side file system tracks open files, and generally translates application requests into the relevant set of protocol messages. The server simply responds to each protocol message, each of which has all the information needed to complete request.

For example, let us consider a simple application which reads a file. In the diagram (Figure 48.5), we show what system calls the application makes, and what the client-side file system and file server do in responding to such calls.

A few comments about the figure. First, notice how the client tracks all relevant **state** for the file access, including the mapping of the integer file descriptor to an NFS file handle as well as the current file pointer. This enables the client to turn each read request (which you may have noticed do *not* specify the offset to read from explicitly) into a properly-formatted read protocol message which tells the server exactly which bytes from the file to read. Upon a successful read, the client updates the current file position; subsequent reads are issued with the same file handle but a different offset.

Second, you may notice where server interactions occur. When the file is opened for the first time, the client-side file system sends a LOOKUP request message. Indeed, if a long pathname must be traversed (e.g., `/home/remzi/foo.txt`), the client would send three LOOKUPs: one to look up `home` in the directory `/`, one to look up `remzi` in `home`, and finally one to look up `foo.txt` in `remzi`.

Third, you may notice how each server request has all the information needed to complete the request in its entirety. This design point is critical to be able to gracefully recover from server failure, as we will now discuss in more detail; it ensures that the server does not need state to be able to respond to the request.

Client	Server
fd = open("/foo", ...);	
Send LOOKUP (rootdir FH, "foo")	
	Receive LOOKUP request look for "foo" in root dir return foo's FH + attributes
Receive LOOKUP reply allocate file desc in open file table store foo's FH in table store current file position (0) return file descriptor to application	
read(fd, buffer, MAX);	
Index into open file table with fd get NFS file handle (FH) use current file position as offset	
Send READ (FH, offset=0, count=MAX)	
	Receive READ request use FH to get volume/inode num read inode from disk (or cache) compute block location (using offset) read data from disk (or cache) return data to client
Receive READ reply update file position (+bytes read) set current file position = MAX return data/error code to app	
read(fd, buffer, MAX);	
Same except offset=MAX and set current file position = 2*MAX	
read(fd, buffer, MAX);	
Same except offset=2*MAX and set current file position = 3*MAX	
close(fd);	
Just need to clean up local structures Free descriptor "fd" in open file table (No need to talk to server)	

Figure 48.5: **Reading A File: Client-side And File Server Actions**

TIP: IDEMPOTENCY IS POWERFUL
Idempotency is a useful property when building reliable systems. When an operation can be issued more than once, it is much easier to handle failure of the operation; you can just retry it. If an operation is *not* idempotent, life becomes more difficult.

48.7 Handling Server Failure with Idempotent Operations

When a client sends a message to the server, it sometimes does not receive a reply. There are many possible reasons for this failure to respond. In some cases, the message may be dropped by the network; networks do lose messages, and thus either the request or the reply could be lost and thus the client would never receive a response.

It is also possible that the server has crashed, and thus is not currently responding to messages. After a bit, the server will be rebooted and start running again, but in the meanwhile all requests have been lost. In all of these cases, clients are left with a question: what should they do when the server does not reply in a timely manner?

In NFSv2, a client handles all of these failures in a single, uniform, and elegant way: it simply *retries* the request. Specifically, after sending the request, the client sets a timer to go off after a specified time period. If a reply is received before the timer goes off, the timer is canceled and all is well. If, however, the timer goes off *before* any reply is received, the client assumes the request has not been processed and resends it. If the server replies, all is well and the client has neatly handled the problem.

The ability of the client to simply retry the request (regardless of what caused the failure) is due to an important property of most NFS requests: they are **idempotent**. An operation is called idempotent when the effect of performing the operation multiple times is equivalent to the effect of performing the operating a single time. For example, if you store a value to a memory location three times, it is the same as doing so once; thus "store value to memory" is an idempotent operation. If, however, you increment a counter three times, it results in a different amount than doing so just once; thus, "increment counter" is not idempotent. More generally, any operation that just reads data is obviously idempotent; an operation that updates data must be more carefully considered to determine if it has this property.

The heart of the design of crash recovery in NFS is the idempotency of most common operations. LOOKUP and READ requests are trivially idempotent, as they only read information from the file server and do not update it. More interestingly, WRITE requests are also idempotent. If, for example, a WRITE fails, the client can simply retry it. The WRITE message contains the data, the count, and (importantly) the exact offset to write the data to. Thus, it can be repeated with the knowledge that the outcome of multiple writes is the same as the outcome of a single one.

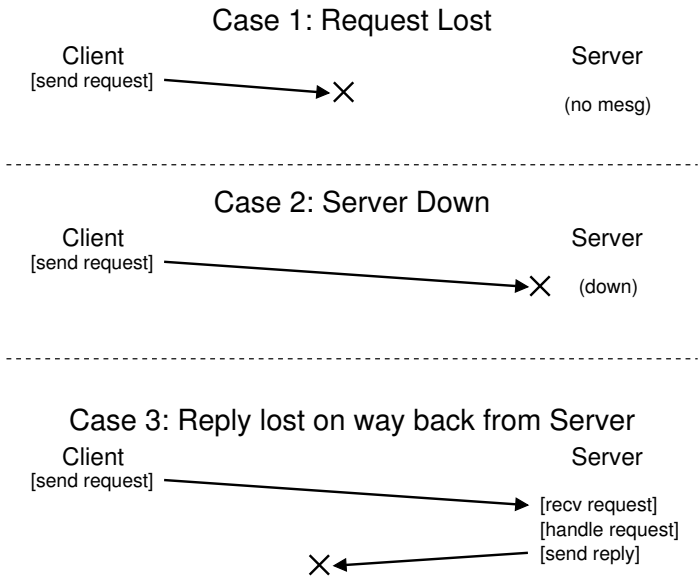

Figure 48.6: **The Three Types of Loss**

In this way, the client can handle all timeouts in a unified way. If a WRITE request was simply lost (Case 1 above), the client will retry it, the server will perform the write, and all will be well. The same will happen if the server happened to be down while the request was sent, but back up and running when the second request is sent, and again all works as desired (Case 2). Finally, the server may in fact receive the WRITE request, issue the write to its disk, and send a reply. This reply may get lost (Case 3), again causing the client to re-send the request. When the server receives the request again, it will simply do the exact same thing: write the data to disk and reply that it has done so. If the client this time receives the reply, all is again well, and thus the client has handled both message loss and server failure in a uniform manner. Neat!

A small aside: some operations are hard to make idempotent. For example, when you try to make a directory that already exists, you are informed that the mkdir request has failed. Thus, in NFS, if the file server receives a MKDIR protocol message and executes it successfully but the reply is lost, the client may repeat it and encounter that failure when in fact the operation at first succeeded and then only failed on the retry. Thus, life is not perfect.

TIP: PERFECT IS THE ENEMY OF THE GOOD (VOLTAIRE'S LAW)
Even when you design a beautiful system, sometimes all the corner cases don't work out exactly as you might like. Take the mkdir example above; one could redesign mkdir to have different semantics, thus making it idempotent (think about how you might do so); however, why bother? The NFS design philosophy covers most of the important cases, and overall makes the system design clean and simple with regards to failure. Thus, accepting that life isn't perfect and still building the system is a sign of good engineering. Apparently, this wisdom is attributed to Voltaire, for saying "... a wise Italian says that the best is the enemy of the good" [V72], and thus we call it **Voltaire's Law**.

48.8 Improving Performance: Client-side Caching

Distributed file systems are good for a number of reasons, but sending all read and write requests across the network can lead to a big performance problem: the network generally isn't that fast, especially as compared to local memory or disk. Thus, another problem: how can we improve the performance of a distributed file system?

The answer, as you might guess from reading the big bold words in the sub-heading above, is client-side **caching**. The NFS client-side file system caches file data (and metadata) that it has read from the server in client memory. Thus, while the first access is expensive (i.e., it requires network communication), subsequent accesses are serviced quite quickly out of client memory.

The cache also serves as a temporary buffer for writes. When a client application first writes to a file, the client buffers the data in client memory (in the same cache as the data it read from the file server) before writing the data out to the server. Such **write buffering** is useful because it decouples application `write()` latency from actual write performance, i.e., the application's call to `write()` succeeds immediately (and just puts the data in the client-side file system's cache); only later does the data get written out to the file server.

Thus, NFS clients cache data and performance is usually great and we are done, right? Unfortunately, not quite. Adding caching into any sort of system with multiple client caches introduces a big and interesting challenge which we will refer to as the **cache consistency problem**.

48.9 The Cache Consistency Problem

The cache consistency problem is best illustrated with two clients and a single server. Imagine client C1 reads a file F, and keeps a copy of the file in its local cache. Now imagine a different client, C2, overwrites the file F, thus changing its contents; let's call the new version of the file F

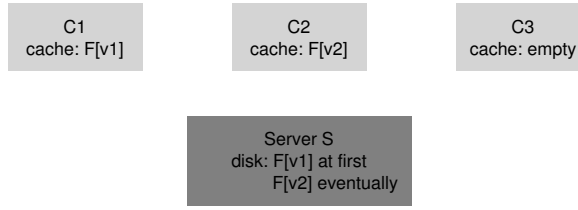

Figure 48.7: **The Cache Consistency Problem**

(version 2), or F[v2] and the old version F[v1] so we can keep the two distinct (but of course the file has the same name, just different contents). Finally, there is a third client, C3, which has not yet accessed the file F.

You can probably see the problem that is upcoming (Figure 48.7). In fact, there are two subproblems. The first subproblem is that the client C2 may buffer its writes in its cache for a time before propagating them to the server; in this case, while F[v2] sits in C2's memory, any access of F from another client (say C3) will fetch the old version of the file (F[v1]). Thus, by buffering writes at the client, other clients may get stale versions of the file, which may be undesirable; indeed, imagine the case where you log into machine C2, update F, and then log into C3 and try to read the file, only to get the old copy! Certainly this could be frustrating. Thus, let us call this aspect of the cache consistency problem **update visibility**; when do updates from one client become visible at other clients?

The second subproblem of cache consistency is a **stale cache**; in this case, C2 has finally flushed its writes to the file server, and thus the server has the latest version (F[v2]). However, C1 still has F[v1] in its cache; if a program running on C1 reads file F, it will get a stale version (F[v1]) and not the most recent copy (F[v2]), which is (often) undesirable.

NFSv2 implementations solve these cache consistency problems in two ways. First, to address update visibility, clients implement what is sometimes called **flush-on-close** (a.k.a., **close-to-open**) consistency semantics; specifically, when a file is written to and subsequently closed by a client application, the client flushes all updates (i.e., dirty pages in the cache) to the server. With flush-on-close consistency, NFS ensures that a subsequent open from another node will see the latest file version.

Second, to address the stale-cache problem, NFSv2 clients first check to see whether a file has changed before using its cached contents. Specifically, when opening a file, the client-side file system will issue a GETATTR request to the server to fetch the file's attributes. The attributes, importantly, include information as to when the file was last modified on the server; if the time-of-modification is more recent than the time that the file was fetched into the client cache, the client **invalidates** the file, thus removing it from the client cache and ensuring that subsequent reads will go to the server and retrieve the latest version of the file. If, on the other

hand, the client sees that it has the latest version of the file, it will go ahead and use the cached contents, thus increasing performance.

When the original team at Sun implemented this solution to the stale-cache problem, they realized a new problem; suddenly, the NFS server was flooded with GETATTR requests. A good engineering principle to follow is to design for the **common case**, and to make it work well; here, although the common case was that a file was accessed only from a single client (perhaps repeatedly), the client always had to send GETATTR requests to the server to make sure no one else had changed the file. A client thus bombards the server, constantly asking "has anyone changed this file?", when most of the time no one had.

To remedy this situation (somewhat), an **attribute cache** was added to each client. A client would still validate a file before accessing it, but most often would just look in the attribute cache to fetch the attributes. The attributes for a particular file were placed in the cache when the file was first accessed, and then would timeout after a certain amount of time (say 3 seconds). Thus, during those three seconds, all file accesses would determine that it was OK to use the cached file and thus do so with no network communication with the server.

48.10 Assessing NFS Cache Consistency

A few final words about NFS cache consistency. The flush-on-close behavior was added to "make sense", but introduced a certain performance problem. Specifically, if a temporary or short-lived file was created on a client and then soon deleted, it would still be forced to the server. A more ideal implementation might keep such short-lived files in memory until they are deleted and thus remove the server interaction entirely, perhaps increasing performance.

More importantly, the addition of an attribute cache into NFS made it very hard to understand or reason about exactly what version of a file one was getting. Sometimes you would get the latest version; sometimes you would get an old version simply because your attribute cache hadn't yet timed out and thus the client was happy to give you what was in client memory. Although this was fine most of the time, it would (and still does!) occasionally lead to odd behavior.

And thus we have described the oddity that is NFS client caching. It serves as an interesting example where details of an implementation serve to define user-observable semantics, instead of the other way around.

48.11 Implications on Server-Side Write Buffering

Our focus so far has been on client caching, and that is where most of the interesting issues arise. However, NFS servers tend to be well-equipped machines with a lot of memory too, and thus they have caching concerns as well. When data (and metadata) is read from disk, NFS

servers will keep it in memory, and subsequent reads of said data (and metadata) will not go to disk, a potential (small) boost in performance.

More intriguing is the case of write buffering. NFS servers absolutely may *not* return success on a WRITE protocol request until the write has been forced to stable storage (e.g., to disk or some other persistent device). While they can place a copy of the data in server memory, returning success to the client on a WRITE protocol request could result in incorrect behavior; can you figure out why?

The answer lies in our assumptions about how clients handle server failure. Imagine the following sequence of writes as issued by a client:

```
write(fd, a_buffer, size); // fill first block with a's
write(fd, b_buffer, size); // fill second block with b's
write(fd, c_buffer, size); // fill third block with c's
```

These writes overwrite the three blocks of a file with a block of a's, then b's, and then c's. Thus, if the file initially looked like this:

```
xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx
yyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyy
zzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzz
```

We might expect the final result after these writes to be like this, with the x's, y's, and z's, would be overwritten with a's, b's, and c's, respectively.

```
aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa
bbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbb
ccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccc
```

Now let's assume for the sake of the example that these three client writes were issued to the server as three distinct WRITE protocol messages. Assume the first WRITE message is received by the server and issued to the disk, and the client informed of its success. Now assume the second write is just buffered in memory, and the server also reports it success to the client *before* forcing it to disk; unfortunately, the server crashes before writing it to disk. The server quickly restarts and receives the third write request, which also succeeds.

Thus, to the client, all the requests succeeded, but we are surprised that the file contents look like this:

```
aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa
yyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyy <--- oops
ccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccc
```

Yikes! Because the server told the client that the second write was successful before committing it to disk, an old chunk is left in the file, which, depending on the application, might be catastrophic.

To avoid this problem, NFS servers *must* commit each write to stable (persistent) storage before informing the client of success; doing so enables the client to detect server failure during a write, and thus retry until

it finally succeeds. Doing so ensures we will never end up with file contents intermingled as in the above example.

The problem that this requirement gives rise to in NFS server implementation is that write performance, without great care, can be *the* major performance bottleneck. Indeed, some companies (e.g., Network Appliance) came into existence with the simple objective of building an NFS server that can perform writes quickly; one trick they use is to first put writes in a battery-backed memory, thus enabling to quickly reply to WRITE requests without fear of losing the data and without the cost of having to write to disk right away; the second trick is to use a file system design specifically designed to write to disk quickly when one finally needs to do so [HLM94, RO91].

48.12 Summary

We have seen the introduction of the NFS distributed file system. NFS is centered around the idea of simple and fast recovery in the face of server failure, and achieves this end through careful protocol design. Idempotency of operations is essential; because a client can safely replay a failed operation, it is OK to do so whether or not the server has executed the request.

We also have seen how the introduction of caching into a multiple-client, single-server system can complicate things. In particular, the system must resolve the cache consistency problem in order to behave reasonably; however, NFS does so in a slightly ad hoc fashion which can occasionally result in observably weird behavior. Finally, we saw how server caching can be tricky: writes to the server must be forced to stable storage before returning success (otherwise data can be lost).

We haven't talked about other issues which are certainly relevant, notably security. Security in early NFS implementations was remarkably lax; it was rather easy for any user on a client to masquerade as other users and thus gain access to virtually any file. Subsequent integration with more serious authentication services (e.g., Kerberos [NT94]) have addressed these obvious deficiencies.

References

[C00] "NFS Illustrated"
Brent Callaghan
Addison-Wesley Professional Computing Series, 2000
A great NFS reference; incredibly thorough and detailed per the protocol itself.

[HLM94] "File System Design for an NFS File Server Appliance"
Dave Hitz, James Lau, Michael Malcolm
USENIX Winter 1994. San Francisco, California, 1994
Hitz et al. were greatly influenced by previous work on log-structured file systems.

[NT94] "Kerberos: An Authentication Service for Computer Networks"
B. Clifford Neuman, Theodore Ts'o
IEEE Communications, 32(9):33-38, September 1994
Kerberos is an early and hugely influential authentication service. We probably should write a book chapter about it sometime...

[O91] "The Role of Distributed State"
John K. Ousterhout
Available: ftp://ftp.cs.berkeley.edu/ucb/sprite/papers/state.ps
A rarely referenced discussion of distributed state; a broader perspective on the problems and challenges.

[P+94] "NFS Version 3: Design and Implementation"
Brian Pawlowski, Chet Juszczak, Peter Staubach, Carl Smith, Diane Lebel, Dave Hitz
USENIX Summer 1994, pages 137-152
The small modifications that underlie NFS version 3.

[P+00] "The NFS version 4 protocol"
Brian Pawlowski, David Noveck, David Robinson, Robert Thurlow
2nd International System Administration and Networking Conference (SANE 2000)
Undoubtedly the most literary paper on NFS ever written.

[RO91] "The Design and Implementation of the Log-structured File System"
Mendel Rosenblum, John Ousterhout
Symposium on Operating Systems Principles (SOSP), 1991
LFS again. No, you can never get enough LFS.

[S86] "The Sun Network File System: Design, Implementation and Experience"
Russel Sandberg
USENIX Summer 1986
The original NFS paper; though a bit of a challenging read, it is worthwhile to see the source of these wonderful ideas.

[Sun89] "NFS: Network File System Protocol Specification"
Sun Microsystems, Inc. Request for Comments: 1094, March 1989
Available: http://www.ietf.org/rfc/rfc1094.txt
The dreaded specification; read it if you must, i.e., you are getting paid to read it. Hopefully, paid a lot. Cash money!

[V72] "La Begueule"
Francois-Marie Arouet a.k.a. Voltaire
Published in 1772
Voltaire said a number of clever things, this being but one example. For example, Voltaire also said "If you have two religions in your land, the two will cut each others throats; but if you have thirty religions, they will dwell in peace." What do you say to that, Democrats and Republicans?

49

The Andrew File System (AFS)

The Andrew File System was introduced by researchers at Carnegie-Mellon University (CMU) in the 1980's [H+88]. Led by the well-known Professor M. Satyanarayanan of Carnegie-Mellon University ("Satya" for short), the main goal of this project was simple: **scale**. Specifically, how can one design a distributed file system such that a server can support as many clients as possible?

Interestingly, there are numerous aspects of design and implementation that affect scalability. Most important is the design of the **protocol** between clients and servers. In NFS, for example, the protocol forces clients to check with the server periodically to determine if cached contents have changed; because each check uses server resources (including CPU and network bandwidth), frequent checks like this will limit the number of clients a server can respond to and thus limit scalability.

AFS also differs from NFS in that from the beginning, reasonable user-visible behavior was a first-class concern. In NFS, cache consistency is hard to describe because it depends directly on low-level implementation details, including client-side cache timeout intervals. In AFS, cache consistency is simple and readily understood: when the file is opened, a client will generally receive the latest consistent copy from the server.

49.1 AFS Version 1

We will discuss two versions of AFS [H+88, S+85]. The first version (which we will call AFSv1, but actually the original system was called the ITC distributed file system [S+85]) had some of the basic design in place, but didn't scale as desired, which led to a re-design and the final protocol (which we will call AFSv2, or just AFS) [H+88]. We now discuss the first version.

One of the basic tenets of all versions of AFS is **whole-file caching** on the **local disk** of the client machine that is accessing a file. When you `open()` a file, the entire file (if it exists) is fetched from the server and stored in a file on your local disk. Subsequent application `read()` and `write()` operations are redirected to the local file system where the file is

```
TestAuth      Test whether a file has changed
              (used to validate cached entries)
GetFileStat   Get the stat info for a file
Fetch         Fetch the contents of file
Store         Store this file on the server
SetFileStat   Set the stat info for a file
ListDir       List the contents of a directory
```

Figure 49.1: **AFSv1 Protocol Highlights**

stored; thus, these operations require no network communication and are fast. Finally, upon `close()`, the file (if it has been modified) is flushed back to the server. Note the obvious contrasts with NFS, which caches *blocks* (not whole files, although NFS could of course cache every block of an entire file) and does so in client *memory* (not local disk).

Let's get into the details a bit more. When a client application first calls `open()`, the AFS client-side code (which the AFS designers call **Venus**) would send a Fetch protocol message to the server. The Fetch protocol message would pass the entire pathname of the desired file (for example, `/home/remzi/notes.txt`) to the file server (the group of which they called **Vice**), which would then traverse the pathname, find the desired file, and ship the entire file back to the client. The client-side code would then cache the file on the local disk of the client (by writing it to local disk). As we said above, subsequent `read()` and `write()` system calls are strictly *local* in AFS (no communication with the server occurs); they are just redirected to the local copy of the file. Because the `read()` and `write()` calls act just like calls to a local file system, once a block is accessed, it also may be cached in client memory. Thus, AFS also uses client memory to cache copies of blocks that it has in its local disk. Finally, when finished, the AFS client checks if the file has been modified (i.e., that it has been opened for writing); if so, it flushes the new version back to the server with a Store protocol message, sending the entire file and pathname to the server for permanent storage.

The next time the file is accessed, AFSv1 does so much more efficiently. Specifically, the client-side code first contacts the server (using the TestAuth protocol message) in order to determine whether the file has changed. If not, the client would use the locally-cached copy, thus improving performance by avoiding a network transfer. The figure above shows some of the protocol messages in AFSv1. Note that this early version of the protocol only cached file contents; directories, for example, were only kept at the server.

49.2 Problems with Version 1

A few key problems with this first version of AFS motivated the designers to rethink their file system. To study the problems in detail, the designers of AFS spent a great deal of time measuring their existing prototype to find what was wrong. Such experimentation is a good thing;

TIP: MEASURE THEN BUILD (PATTERSON'S LAW)
One of our advisors, David Patterson (of RISC and RAID fame), used to always encourage us to measure a system and demonstrate a problem *before* building a new system to fix said problem. By using experimental evidence, rather than gut instinct, you can turn the process of system building into a more scientific endeavor. Doing so also has the fringe benefit of making you think about how exactly to measure the system before your improved version is developed. When you do finally get around to building the new system, two things are better as a result: first, you have evidence that shows you are solving a real problem; second, you now have a way to measure your new system in place, to show that it actually improves upon the state of the art. And thus we call this **Patterson's Law**.

measurement is the key to understanding how systems work and how to improve them. Hard data helps take intuition and make into a concrete science of deconstructing systems. In their study, the authors found two main problems with AFSv1:

- **Path-traversal costs are too high**: When performing a Fetch or Store protocol request, the client passes the entire pathname (e.g., `/home/remzi/notes.txt`) to the server. The server, in order to access the file, must perform a full pathname traversal, first looking in the root directory to find `home`, then in `home` to find `remzi`, and so forth, all the way down the path until finally the desired file is located. With many clients accessing the server at once, the designers of AFS found that the server was spending much of its CPU time simply walking down directory paths.
- **The client issues too many TestAuth protocol messages**: Much like NFS and its overabundance of GETATTR protocol messages, AFSv1 generated a large amount of traffic to check whether a local file (or its stat information) was valid with the TestAuth protocol message. Thus, servers spent much of their time telling clients whether it was OK to used their cached copies of a file. Most of the time, the answer was that the file had not changed.

There were actually two other problems with AFSv1: load was not balanced across servers, and the server used a single distinct process per client thus inducing context switching and other overheads. The load imbalance problem was solved by introducing **volumes**, which an administrator could move across servers to balance load; the context-switch problem was solved in AFSv2 by building the server with threads instead of processes. However, for the sake of space, we focus here on the main two protocol problems above that limited the scale of the system.

49.3 Improving the Protocol

The two problems above limited the scalability of AFS; the server CPU became the bottleneck of the system, and each server could only service 20 clients without becoming overloaded. Servers were receiving too many TestAuth messages, and when they received Fetch or Store messages, were spending too much time traversing the directory hierarchy. Thus, the AFS designers were faced with a problem:

THE CRUX: HOW TO DESIGN A SCALABLE FILE PROTOCOL
How should one redesign the protocol to minimize the number of server interactions, i.e., how could they reduce the number of TestAuth messages? Further, how could they design the protocol to make these server interactions efficient? By attacking both of these issues, a new protocol would result in a much more scalable version AFS.

49.4 AFS Version 2

AFSv2 introduced the notion of a **callback** to reduce the number of client/server interactions. A callback is simply a promise from the server to the client that the server will inform the client when a file that the client is caching has been modified. By adding this **state** to the server, the client no longer needs to contact the server to find out if a cached file is still valid. Rather, it assumes that the file is valid until the server tells it otherwise; insert analogy to **polling** versus **interrupts** here.

AFSv2 also introduced the notion of a **file identifier** (**FID**) (similar to the NFS **file handle**) instead of pathnames to specify which file a client was interested in. An FID in AFS consists of a volume identifier, a file identifier, and a "uniquifier" (to enable reuse of the volume and file IDs when a file is deleted). Thus, instead of sending whole pathnames to the server and letting the server walk the pathname to find the desired file, the client would walk the pathname, one piece at a time, caching the results and thus hopefully reducing the load on the server.

For example, if a client accessed the file `/home/remzi/notes.txt`, and `home` was the AFS directory mounted onto `/` (i.e., `/` was the local root directory, but `home` and its children were in AFS), the client would first Fetch the directory contents of `home`, put them in the local-disk cache, and setup a callback on `home`. Then, the client would Fetch the directory `remzi`, put it in the local-disk cache, and setup a callback on the server on `remzi`. Finally, the client would Fetch `notes.txt`, cache this regular file in the local disk, setup a callback, and finally return a file descriptor to the calling application. See Figure 49.2 for a summary.

The key difference, however, from NFS, is that with each fetch of a directory or file, the AFS client would establish a callback with the server, thus ensuring that the server would notify the client of a change in its

Client (C_1)	Server
fd = open("/home/remzi/notes.txt", ...);	
Send Fetch (home FID, "remzi")	
	Receive Fetch request look for remzi in home dir establish callback(C_1) on remzi return remzi's content and FID
Receive Fetch reply write remzi to local disk cache record callback status of remzi Send Fetch (remzi FID, "notes.txt")	
	Receive Fetch request look for notes.txt in remzi dir establish callback(C_1) on notes.txt return notes.txt's content and FID
Receive Fetch reply write notes.txt to local disk cache record callback status of notes.txt local `open()` of cached notes.txt return file descriptor to application	
read(fd, buffer, MAX); perform local `read()` on cached copy	
close(fd); do local `close()` on cached copy if file has changed, flush to server	

fd = open("/home/remzi/notes.txt", ...);
Foreach dir (home, remzi)
if (callback(dir) == VALID)
use local copy for lookup(dir)
else
Fetch (as above)
if (callback(notes.txt) == VALID)
open local cached copy
return file descriptor to it
else
Fetch (as above) then open and return fd	

Figure 49.2: **Reading A File: Client-side And File Server Actions**

cached state. The benefit is obvious: although the *first* access to `/home/remzi/notes.txt` generates many client-server messages (as described above), it also establishes callbacks for all the directories as well as the file notes.txt, and thus subsequent accesses are entirely local and require no server interaction at all. Thus, in the common case where a file is cached at the client, AFS behaves nearly identically to a local disk-based file system. If one accesses a file more than once, the second access should be just as fast as accessing a file locally.

ASIDE: **CACHE CONSISTENCY IS NOT A PANACEA**
When discussing distributed file systems, much is made of the cache consistency the file systems provide. However, this baseline consistency does not solve all problems with regards to file access from multiple clients. For example, if you are building a code repository, with multiple clients performing check-ins and check-outs of code, you can't simply rely on the underlying file system to do all of the work for you; rather, you have to use explicit **file-level locking** in order to ensure that the "right" thing happens when such concurrent accesses take place. Indeed, any application that truly cares about concurrent updates will add extra machinery to handle conflicts. The baseline consistency described in this chapter and the previous one are useful primarily for casual usage, i.e., when a user logs into a different client, they expect some reasonable version of their files to show up there. Expecting more from these protocols is setting yourself up for failure, disappointment, and tear-filled frustration.

49.5 Cache Consistency

When we discussed NFS, there were two aspects of cache consistency we considered: **update visibility** and **cache staleness**. With update visibility, the question is: when will the server be updated with a new version of a file? With cache staleness, the question is: once the server has a new version, how long before clients see the new version instead of an older cached copy?

Because of callbacks and whole-file caching, the cache consistency provided by AFS is easy to describe and understand. There are two important cases to consider: consistency between processes on *different* machines, and consistency between processes on the *same* machine.

Between different machines, AFS makes updates visible at the server and invalidates cached copies at the exact same time, which is when the updated file is closed. A client opens a file, and then writes to it (perhaps repeatedly). When it is finally closed, the new file is flushed to the server (and thus visibile); the server then breaks callbacks for any clients with cached copies, thus ensuring that clients will no longer read stale copies of the file; subsequent opens on those clients will require a re-fetch of the new version of the file from the server.

AFS makes an exception to this simple model between processes on the same machine. In this case, writes to a file are immediately visible to other local processes (i.e., a process does not have to wait until a file is closed to see its latest updates). This makes using a single machine behave exactly as you would expect, as this behavior is based upon typical UNIX semantics. Only when switching to a different machine would you be able to detect the more general AFS consistency mechanism.

There is one interesting cross-machine case that is worthy of further discussion. Specifically, in the rare case that processes on different ma-

$Client_1$			$Client_2$		Server	*Comments*
P_1	P_2	Cache	P_3	Cache	Disk	
open(F)		-		-	-	File created
write(A)		A		-	-	
close()		A		-	A	
	open(F)	A		-	A	
	read() → A	A		-	A	
	close()	A		-	A	
open(F)		A		-	A	
write(B)		B		-	A	
	open(F)	B		-	A	Local processes
	read() → B	B		-	A	see writes immediately
	close()	B		-	A	
		B	open(F)	A	A	Remote processes
		B	read() → A	A	A	do not see writes...
		B	close()	A	A	
close()		B		~~A~~	B	... until close()
		B	open(F)	B	B	has taken place
		B	read() → B	B	B	
		B	close()	B	B	
		B	open(F)	B	B	
open(F)		B		B	B	
write(D)		D		B	B	
		D	write(C)	C	B	
		D	close()	C	C	
close()		D		~~C~~	D	
		D	open(F)	D	D	Unfortunately for P_3
		D	read() → D	D	D	the last writer wins
		D	close()	D	D	

Figure 49.3: **Cache Consistency Timeline**

chines are modifying a file at the same time, AFS naturally employs what is known as a **last writer wins** approach (which perhaps should be called **last closer wins**). Specifically, whichever client calls `close()` last will update the entire file on the server last and thus will be the "winning" file, i.e., the file that remains on the server for others to see. The result is a file that was generated in its entirety either by one client or the other. Note the difference from a block-based protocol like NFS: in NFS, writes of individual blocks may be flushed out to the server as each client is updating the file, and thus the final file on the server could end up as a mix of updates from both clients. In many cases, such a mixed file output would not make much sense, i.e., imagine a JPEG image getting modified by two clients in pieces; the resulting mix of writes would not likely constitute a valid JPEG.

A timeline showing a few of these different scenarios can be seen in Figure 49.3. The columns show the behavior of two processes (P_1 and P_2) on $Client_1$ and its cache state, one process (P_3) on $Client_2$ and its cache state, and the server (Server), all operating on a single file called, imaginatively, `F`. For the server, the figure simply shows the contents of the file after the operation on the left has completed. Read through it and see if you can understand why each read returns the results that it does. A commentary field on the right will help you if you get stuck.

49.6 Crash Recovery

From the description above, you might sense that crash recovery is more involved than with NFS. You would be right. For example, imagine there is a short period of time where a server (S) is not able to contact a client (C1), for example, while the client C1 is rebooting. While C1 is not available, S may have tried to send it one or more callback recall messages; for example, imagine C1 had file F cached on its local disk, and then C2 (another client) updated F, thus causing S to send messages to all clients caching the file to remove it from their local caches. Because C1 may miss those critical messages when it is rebooting, upon rejoining the system, C1 should treat all of its cache contents as suspect. Thus, upon the next access to file F, C1 should first ask the server (with a TestAuth protocol message) whether its cached copy of file F is still valid; if so, C1 can use it; if not, C1 should fetch the newer version from the server.

Server recovery after a crash is also more complicated. The problem that arises is that callbacks are kept in memory; thus, when a server reboots, it has no idea which client machine has which files. Thus, upon server restart, each client of the server must realize that the server has crashed and treat all of their cache contents as suspect, and (as above) reestablish the validity of a file before using it. Thus, a server crash is a big event, as one must ensure that each client is aware of the crash in a timely manner, or risk a client accessing a stale file. There are many ways to implement such recovery; for example, by having the server send a message (saying "don't trust your cache contents!") to each client when it is up and running again, or by having clients check that the server is alive periodically (with a **heartbeat** message, as it is called). As you can see, there is a cost to building a more scalable and sensible caching model; with NFS, clients hardly noticed a server crash.

49.7 Scale And Performance Of AFSv2

With the new protocol in place, AFSv2 was measured and found to be much more scalable that the original version. Indeed, each server could support about 50 clients (instead of just 20). A further benefit was that client-side performance often came quite close to local performance, because in the common case, all file accesses were local; file reads usually went to the local disk cache (and potentially, local memory). Only when a client created a new file or wrote to an existing one was there need to send a Store message to the server and thus update the file with new contents.

Let us also gain some perspective on AFS performance by comparing common file-system access scenarios with NFS. Figure 49.4 (page 9) shows the results of our qualitative comparison.

In the figure, we examine typical read and write patterns analytically, for files of different sizes. Small files have N_s blocks in them; medium files have N_m blocks; large files have N_L blocks. We assume that small

Workload	NFS	AFS	AFS/NFS
1. Small file, sequential read	$N_s \cdot L_{net}$	$N_s \cdot L_{net}$	1
2. Small file, sequential re-read	$N_s \cdot L_{mem}$	$N_s \cdot L_{mem}$	1
3. Medium file, sequential read	$N_m \cdot L_{net}$	$N_m \cdot L_{net}$	1
4. Medium file, sequential re-read	$N_m \cdot L_{mem}$	$N_m \cdot L_{mem}$	1
5. Large file, sequential read	$N_L \cdot L_{net}$	$N_L \cdot L_{net}$	1
6. Large file, sequential re-read	$N_L \cdot L_{net}$	$N_L \cdot L_{disk}$	$\frac{L_{disk}}{L_{net}}$
7. Large file, single read	L_{net}	$N_L \cdot L_{net}$	N_L
8. Small file, sequential write	$N_s \cdot L_{net}$	$N_s \cdot L_{net}$	1
9. Large file, sequential write	$N_L \cdot L_{net}$	$N_L \cdot L_{net}$	1
10. Large file, sequential overwrite	$N_L \cdot L_{net}$	$2 \cdot N_L \cdot L_{net}$	2
11. Large file, single write	L_{net}	$2 \cdot N_L \cdot L_{net}$	$2 \cdot N_L$

Figure 49.4: **Comparison: AFS vs. NFS**

and medium files fit into the memory of a client; large files fit on a local disk but not in client memory.

We also assume, for the sake of analysis, that an access across the network to the remote server for a file block takes L_{net} time units. Access to local memory takes L_{mem}, and access to local disk takes L_{disk}. The general assumption is that $L_{net} > L_{disk} > L_{mem}$.

Finally, we assume that the first access to a file does not hit in any caches. Subsequent file accesses (i.e., "re-reads") we assume will hit in caches, if the relevant cache has enough capacity to hold the file.

The columns of the figure show the time a particular operation (e.g., a small file sequential read) roughly takes on either NFS or AFS. The rightmost column displays the ratio of AFS to NFS.

We make the following observations. First, in many cases, the performance of each system is roughly equivalent. For example, when first reading a file (e.g., Workloads 1, 3, 5), the time to fetch the file from the remote server dominates, and is similar on both systems. You might think AFS would be slower in this case, as it has to write the file to local disk; however, those writes are buffered by the local (client-side) file system cache and thus said costs are likely hidden. Similarly, you might think that AFS reads from the local cached copy would be slower, again because AFS stores the cached copy on disk. However, AFS again benefits here from local file system caching; reads on AFS would likely hit in the client-side memory cache, and performance would be similar to NFS.

Second, an interesting difference arises during a large-file sequential re-read (Workload 6). Because AFS has a large local disk cache, it will access the file from there when the file is accessed again. NFS, in contrast, only can cache blocks in client memory; as a result, if a large file (i.e., a file bigger than local memory) is re-read, the NFS client will have to re-fetch the entire file from the remote server. Thus, AFS is faster than NFS in this case by a factor of $\frac{L_{net}}{L_{disk}}$, assuming that remote access is indeed slower than local disk. We also note that NFS in this case increases server load, which has an impact on scale as well.

Third, we note that sequential writes (of new files) should perform similarly on both systems (Workloads 8, 9). AFS, in this case, will write the file to the local cached copy; when the file is closed, the AFS client will force the writes to the server, as per the protocol. NFS will buffer writes in client memory, perhaps forcing some blocks to the server due to client-side memory pressure, but definitely writing them to the server when the file is closed, to preserve NFS flush-on-close consistency. You might think AFS would be slower here, because it writes all data to local disk. However, realize that it is writing to a local file system; those writes are first committed to the page cache, and only later (in the background) to disk, and thus AFS reaps the benefits of the client-side OS memory caching infrastructure to improve performance.

Fourth, we note that AFS performs worse on a sequential file overwrite (Workload 10). Thus far, we have assumed that the workloads that write are also creating a new file; in this case, the file exists, and is then over-written. Overwrite can be a particularly bad case for AFS, because the client first fetches the old file in its entirety, only to subsequently overwrite it. NFS, in contrast, will simply overwrite blocks and thus avoid the initial (useless) read[1].

Finally, workloads that access a small subset of data within large files perform much better on NFS than AFS (Workloads 7, 11). In these cases, the AFS protocol fetches the entire file when the file is opened; unfortunately, only a small read or write is performed. Even worse, if the file is modified, the entire file is written back to the server, doubling the performance impact. NFS, as a block-based protocol, performs I/O that is proportional to the size of the read or write.

Overall, we see that NFS and AFS make different assumptions and not surprisingly realize different performance outcomes as a result. Whether these differences matter is, as always, a question of workload.

49.8 AFS: Other Improvements

Like we saw with the introduction of Berkeley FFS (which added symbolic links and a number of other features), the designers of AFS took the opportunity when building their system to add a number of features that made the system easier to use and manage. For example, AFS provides a true global namespace to clients, thus ensuring that all files were named the same way on all client machines. NFS, in contrast, allows each client to mount NFS servers in any way that they please, and thus only by convention (and great administrative effort) would files be named similarly across clients.

[1] We assume here that NFS reads are block-sized and block-aligned; if they were not, the NFS client would also have to read the block first. We also assume the file was *not* opened with the O_TRUNC flag; if it had been, the initial open in AFS would not fetch the soon to be truncated file's contents.

ASIDE: **THE IMPORTANCE OF WORKLOAD**

One challenge of evaluating any system is the choice of **workload**. Because computer systems are used in so many different ways, there are a large variety of workloads to choose from. How should the storage system designer decide which workloads are important, in order to make reasonable design decisions?

The designers of AFS, given their experience in measuring how file systems were used, made certain workload assumptions; in particular, they assumed that most files were not frequently shared, and accessed sequentially in their entirety. Given those assumptions, the AFS design makes perfect sense.

However, these assumptions are not always correct. For example, imagine an application that appends information, periodically, to a log. These little log writes, which add small amounts of data to an existing large file, are quite problematic for AFS. Many other difficult workloads exist as well, e.g., random updates in a transaction database.

One place to get some information about what types of workloads are common are through various research studies that have been performed. See any of these studies for good examples of workload analysis [B+91, H+11, R+00, V99], including the AFS retrospective [H+88].

AFS also takes security seriously, and incorporates mechanisms to authenticate users and ensure that a set of files could be kept private if a user so desired. NFS, in contrast, had quite primitive support for security for many years.

AFS also includes facilities for flexible user-managed access control. Thus, when using AFS, a user has a great deal of control over who exactly can access which files. NFS, like most UNIX file systems, has much less support for this type of sharing.

Finally, as mentioned before, AFS adds tools to enable simpler management of servers for the administrators of the system. In thinking about system management, AFS was light years ahead of the field.

49.9 Summary

AFS shows us how distributed file systems can be built quite differently than what we saw with NFS. The protocol design of AFS is particularly important; by minimizing server interactions (through whole-file caching and callbacks), each server can support many clients and thus reduce the number of servers needed to manage a particular site. Many other features, including the single namespace, security, and access-control lists, make AFS quite nice to use. The consistency model provided by AFS is simple to understand and reason about, and does not lead to the occasional weird behavior as one sometimes observes in NFS.

Perhaps unfortunately, AFS is likely on the decline. Because NFS became an open standard, many different vendors supported it, and, along with CIFS (the Windows-based distributed file system protocol), NFS dominates the marketplace. Although one still sees AFS installations from time to time (such as in various educational institutions, including Wisconsin), the only lasting influence will likely be from the ideas of AFS rather than the actual system itself. Indeed, NFSv4 now adds server state (e.g., an "open" protocol message), and thus bears an increasing similarity to the basic AFS protocol.

References

[B+91] "Measurements of a Distributed File System"
Mary Baker, John Hartman, Martin Kupfer, Ken Shirriff, John Ousterhout
SOSP '91, Pacific Grove, California, October 1991
An early paper measuring how people use distributed file systems. Matches much of the intuition found in AFS.

[H+11] "A File is Not a File: Understanding the I/O Behavior of Apple Desktop Applications"
Tyler Harter, Chris Dragga, Michael Vaughn,
Andrea C. Arpaci-Dusseau, Remzi H. Arpaci-Dusseau
SOSP '11, New York, New York, October 2011
Our own paper studying the behavior of Apple Desktop workloads; turns out they are a bit different than many of the server-based workloads the systems research community usually focuses upon. Also a good recent reference which points to a lot of related work.

[H+88] "Scale and Performance in a Distributed File System"
John H. Howard, Michael L. Kazar, Sherri G. Menees, David A. Nichols, M. Satyanarayanan, Robert N. Sidebotham, Michael J. West
ACM Transactions on Computing Systems (ACM TOCS), page 51-81, Volume 6, Number 1, February 1988
The long journal version of the famous AFS system, still in use in a number of places throughout the world, and also probably the earliest clear thinking on how to build distributed file systems. A wonderful combination of the science of measurement and principled engineering.

[R+00] "A Comparison of File System Workloads"
Drew Roselli, Jacob R. Lorch, Thomas E. Anderson
USENIX '00, San Diego, California, June 2000
A more recent set of traces as compared to the Baker paper [B+91], with some interesting twists.

[S+85] "The ITC Distributed File System: Principles and Design"
M. Satyanarayanan, J.H. Howard, D.A. Nichols, R.N. Sidebotham, A. Spector, M.J. West
SOSP '85, Orcas Island, Washington, December 1985
The older paper about a distributed file system. Much of the basic design of AFS is in place in this older system, but not the improvements for scale.

[V99] "File system usage in Windows NT 4.0"
Werner Vogels
SOSP '99, Kiawah Island Resort, South Carolina, December 1999
A cool study of Windows workloads, which are inherently different than many of the UNIX*-based studies that had previously been done.*

Homework

This section introduces `afs.py`, a simple AFS simulator you can use to shore up your knowledge of how the Andrew File System works. Read the README file for more details.

Questions

1. Run a few simple cases to make sure you can predict what values will be read by clients. Vary the random seed flag (`-s`) and see if you can trace through and predict both intermediate values as well as the final values stored in the files. Also vary the number of files (`-f`), the number of clients (`-C`), and the read ratio (`-r`, from between 0 to 1) to make it a bit more challenging. You might also want to generate slightly longer traces to make for more interesting interactions, e.g., (`-n 2` or higher).
2. Now do the same thing and see if you can predict each callback that the AFS server initiates. Try different random seeds, and make sure to use a high level of detailed feedback (e.g., `-d 3`) to see when callbacks occur when you have the program compute the answers for you (with `-c`). Can you guess exactly when each callback occurs? What is the precise condition for one to take place?
3. Similar to above, run with some different random seeds and see if you can predict the exact cache state at each step. Cache state can be observed by running with `-c` and `-d 7`.
4. Now let's construct some specific workloads. Run the simulation with `-A oa1:w1:c1,oa1:r1:c1` flag. What are different possible values observed by client 1 when it reads the file `a`, when running with the random scheduler? (try different random seeds to see different outcomes)? Of all the possible schedule interleavings of the two clients' operations, how many of them lead to client 1 reading the value 1, and how many reading the value 0?
5. Now let's construct some specific schedules. When running with the `-A oa1:w1:c1,oa1:r1:c1` flag, also run with the following schedules: `-S 01`, `-S 100011`, `-S 011100`, and others of which you can think. What value will client 1 read?
6. Now run with this workload: `-A oa1:w1:c1,oa1:w1:c1`, and vary the schedules as above. What happens when you run with `-S 011100`? What about when you run with `-S 010011`? What is important in determining the final value of the file?

50

Summary Dialogue on Distribution

Student: *Well, that was quick. Too quick, in my opinion!*

Professor: *Yes, distributed systems are complicated and cool and well worth your study; just not in this book (or course).*

Student: *That's too bad; I wanted to learn more! But I did learn a few things.*

Professor: *Like what?*

Student: *Well, everything can fail.*

Professor: *Good start.*

Student: *But by having lots of these things (whether disks, machines, or whatever), you can hide much of the failure that arises.*

Professor: *Keep going!*

Student: *Some basic techniques like retrying are really useful.*

Professor: *That's true.*

Student: *And you have to think carefully about protocols: the exact bits that are exchanged between machines. Protocols can affect everything, including how systems respond to failure and how scalable they are.*

Professor: *You really are getting better at this learning stuff.*

Student: *Thanks! And you're not a bad teacher yourself!*

Professor: *Well thank you very much too.*

Student: *So is this the end of the book?*

Professor: *I'm not sure. They don't tell me anything.*

Student: *Me neither. Let's get out of here.*

Professor: *OK.*

Student: *Go ahead.*

Professor: *No, after you.*

Student: *Please, professors first.*

Professor: *No, please, after you.*

Student: *(exasperated) Fine!*

Professor: *(waiting) ... so why haven't you left?*

Student: *I don't know how. Turns out, the only thing I can do is participate in these dialogues.*

Professor: *Me too. And now you've learned our final lesson...*