Remote Procedure Call Protocols
The basic theory of operation of RPC is pretty straightforward. But, to understand remote procedure calls, let’s first make sure that we understand local procedure calls. The client (or caller) supplies some number of arguments and invokes the procedure. The server (or callee) receives the invocation and gets a copy of the arguments (other languages, such as C++, provide other argument-passing modes, but copying is all that is provided in C). In the usual implementation, the callee’s copy of the arguments have been placed on the runtime stack by the caller — the callee code knows exactly where to find them. When the call completes, a return value may be supplied by the callee to the caller. Some of the arguments might be out arguments — changes to their value are reflected back to the caller. This is handled in C indirectly — the actual argument, passed by copying, is a pointer to some value. The callee follows the pointer and modifies the value.
Now suppose that the client and server are on separate machines. As much as possible, we would like remote procedure calling to look and behave like local procedure calling. Furthermore, we would like to use the same languages and compilers for the remote case as in the local case. But how do we make this work? A remote call is very different from a local call. For example, in the local call, the caller simply puts the arguments on the runtime stack and expects the callee to find them there. In C, the callee returns data through out arguments by following a pointer into the space of the caller. These techniques simply don’t work in the remote case.
The solution is to use stub procedures: the client places a call to something that has the name of the desired procedure, but is actually a proxy for it, known as the client-side stub. This proxy gathers together all of the arguments (actually, just the in and in-out arguments) and packages them into a message that it sends to the server. The server has a corresponding server-side stub that receives the invocation message, pulls out the arguments, and calls the actual (remote) procedure. When this procedure returns, returned data is packaged by the server-side stub into another message, which is transmitted back to the client-side stub, which pulls out the data and returns it to the original caller. From the points of view of the caller and callee procedure, the entire process appears to be a local procedure call — they behave no differently for the remote case.
ONC RPC

- Used with NFS
- eXternal Data Representation (XDR)
  - specification for how data is transmitted
  - language for specifying interfaces
Example

typedef struct {
    int   comp1;
    float comp2[6];
    char  *annotation;
} value_t;

typedef struct {
    value_t   item;
    list_t    *next;
} list_t;

bool add(int key, value_t item);
bool remove(int key, value_t item);
list_t query(int key);
Placing a Call

```cpp
bool add(int k, value_t v) {
    ...
    return result;
}
```

result = add(key, item);
Returning From the Call

```
result = add(key, item);
```

```
char add(int k, value_t v) {
    ...
    return(result);
}
```

Diagram:
- marshal
- unmarshal
- Wire
### Marshalled Arguments

<table>
<thead>
<tr>
<th>int</th>
<th>key</th>
</tr>
</thead>
<tbody>
<tr>
<td>int</td>
<td>comp1</td>
</tr>
<tr>
<td>float (1)</td>
<td>comp2</td>
</tr>
<tr>
<td>float (2)</td>
<td>item</td>
</tr>
<tr>
<td>float (3)</td>
<td>annotation</td>
</tr>
<tr>
<td>float (4)</td>
<td></td>
</tr>
<tr>
<td>float (5)</td>
<td></td>
</tr>
<tr>
<td>float (6)</td>
<td></td>
</tr>
<tr>
<td>string length</td>
<td></td>
</tr>
<tr>
<td>string (1 – 4)</td>
<td></td>
</tr>
<tr>
<td>string (5 – 8)</td>
<td></td>
</tr>
<tr>
<td>...</td>
<td></td>
</tr>
</tbody>
</table>
### Marshalled Linked List

<table>
<thead>
<tr>
<th></th>
<th>array length</th>
</tr>
</thead>
<tbody>
<tr>
<td>0:</td>
<td>value_t</td>
</tr>
<tr>
<td>1:</td>
<td>value_t</td>
</tr>
<tr>
<td>2:</td>
<td>value_t</td>
</tr>
<tr>
<td>3:</td>
<td>value_t</td>
</tr>
<tr>
<td>4:</td>
<td>value_t</td>
</tr>
<tr>
<td>5:</td>
<td>value_t</td>
</tr>
<tr>
<td>6:</td>
<td>value_t</td>
</tr>
</tbody>
</table>
Reliability Explained …

- Exactly once semantics
  - each RPC request is handled exactly once on the server
Note that if these exchanges are done over TCP, then each request and each response requires two packets — the message and then the ack. In UDP the response can be thought of as acknowledging the request, and the next request as acknowledging the response. Thus, in principle, RPC layered on UDP requires fewer messages than RPC layered on TCP.
Consider this slide first with the assumption that RPC is layered on UDP. Thus, since the response acts as the acknowledgement, there is uncertainty as to whether the request was handled by the server.

Does this uncertainty go away if RPC is layered on TCP? If you consider the possibility that the TCP connection might be lost, perhaps due to a transient network problem, the answer is clearly no. For example, suppose the TCP connection is lost just after the server receives the request. With no connection, the server cannot send a response, so the client is uncertain about what happened.
A procedure is *idempotent* if the effect of executing it twice in a row is the same as executing it just once. With such procedures, the client may repeatedly send a request until it finally gets a response. If an RPC protocol depends on such retries, it is said to have *at-least-once semantics* — clients are assured that, after all the retries, the remote procedure is executed at least once.
Not everything is idempotent! If we have non-idempotent procedures, then RPC requests should not be blindly retried, but instead should be sent just once. RPC protocols that do this are said to have at-most-once semantics.
The server might keep track of what operations it has already performed and what the responses were. If it gets a repeat of a previous request, it merely repeats the original response.
If the server crashed and no longer has its history information, it can respond by raising an exception at the client, indicating that it has no knowledge as to whether the operation has taken place. But it guarantees that it hasn’t taken place more than once.
Making ONC RPC Reliable

- Each request uniquely identified by *transmission ID* (XID)
  - transmission and retransmission share same XID
- Server maintains *duplicate request cache* (DRC)
  - holds XIDs of non-idempotent requests and copies of their complete responses
  - kept in cache for a few minutes
Note that idempotent requests are not cached and hence won’t be recognized as duplicates.
Did It Work?

- No
Problem ...

Client → write(data) xid=1 → nfsd1

Client → write(data) xid=1 (retransmission) → nfsd2

Client → done → updates file

Client → write(newdata) xid=2 → nfsd2

Client → done → updates file

nfsd2 → done → updates file

blocks → done → updates file
Did It Work?

• Sort of ...
• Works fine in well behaved networks
• Doesn’t work with “Byzantine” routers
  – programmed by your worst (and brightest) enemy
  – probably doesn't occur in local environment
  – good approximation of behavior on overloaded Internet
• Doesn’t work if server crashes at inopportune moment (and comes back up)
Enter TCP

- Any improvement?
  - no ...
What’s Wrong?

- The problem is the duplicate request cache (DRC)
  - it’s necessary
  - but when may cached entries be removed?
In session-oriented RPC, the client, as part of setting up a session with the server, lets the server know ahead of time the maximum number of concurrent requests it will issue; the server then creates that number of channels for it. Each client request then provides a channel number and a sequence number. Only one request at a time may be active on a channel. Requests on each channel carry consecutive sequence numbers. The server maintains a separate DRC entry for each channel of each client, containing the XID, sequence number, and response, as shown in the slide. An entry’s contents are not deleted until a request arrives with the next sequence number for that channel.
To be precise, Digital Equipment Corporation (DEC) was purchased by Compaq in 1998, which was acquired by HP in 2002. Apollo was acquired by HP in 1989.

DCE RPC

- Designed by Apollo and Digital in the 1980s
  – both companies later absorbed by HP
- Does everything ONC RPC can do, and more
- Basis for Microsoft RPC
Now we re-do the example using DCE RPC.

typedef struct {
    double comp1;
    int comp2;
    long long comp3;
    char *annotation;
} value_t;

char add(int key, value_t value);
char remove(int key, value_t value);
int query(int key, int number, value_t values[]);
DCE RPC uses an augmented C syntax known as *interface definition language* (IDL) to express its interfaces. By compiling a description written in IDL (using a special IDL compiler), one automatically produces the client- and server-side stubs.

Here we have the IDL description for the simple database example of the previous slide. It starts with the declaration of a data type that is that of the somewhat complex items stored in the database. Following this are the specifications of the three (remote) procedures that clients may call — one for adding values to the database, one for removing values, and one for issuing queries. Values are entered into the database in association with keys. The query operation returns all values, up to an indicated maximum number, that share the given key.

One of the purposes of IDL is to overcome various shortcomings of the C syntax for declaring procedures. Among the issues are:

- Which arguments are input arguments, which are output arguments, and which are both? There is no way to determine this from standard C syntax. With IDL, we have new attributes, enclosed in square brackets, that identify the use of the arguments.

- What is an integer? C has three signed integer types: char, short, and long. One can also declare something to be an int, but, depending on the architecture, it will either be a short or a long. (Of course, we must at some point deal with 64-bit architectures. IDL has a 64-bit integer type called hyper.) Short and long are pretty straightforward, but what is a char? Its name certainly implies some ambiguity. To eliminate this ambiguity, in IDL, if one wants an 8-bit signed integer, one calls it a small.
• If we pass an array, how big is it? From our knowledge of the program, we can see that one of the arguments is the size of the two arrays, but how would the IDL compiler know this? It must know the size of the arrays, so that it can determine how much data to pass to the server (and how much data the server should pass to the client for output arguments). In the example here, we use the `size_is` attribute to notify the IDL compiler that one of the arguments is the size of the array.

• What does it mean to pass a pointer argument? For example, if one of the arguments is declared `char *` what would that mean? We certainly don't want to follow a pointer on the server back to the client, so we should pass the data pointed to along with the pointer. But how much should be passed? Is a `char *` really a pointer to a single character? Is it a pointer to a null-terminated string? Is it a pointer to a counted array of bytes? The IDL syntax allows us to distinguish these cases (and supply whatever additional information is necessary).

• When we pass strings of characters, which character set are we referring to? ASCII is only acceptable (and just barely) in English-speaking countries (for which a seven-bit character set suffices). In the example we specify ISO_LATIN_1, an eight-bit character set, which is suitable for most of Europe and the Americas, but for little of Asia, which needs a sixteen-bit (at least) character set.

There are alternatives to this approach to handling the data types of parameters. One popular approach, used in a early RPC protocol developed at Xerox in the '70s and now popularized in Microsoft’s .Net and other recent systems, is, rather than have linked-in stubs that “know” the types of parameters, to send the typing information along with the parameters. Thus, rather than simply sending the integer “6”, what would be sent is “INT 6”. This is currently being done in conjunction with XML.

The `idempotent` attribute on `query` indicates that `query` is idempotent, and thus it can be retried in the event of no response. Procedures without this attribute are assumed not
To represent an array, we need to include its length.
Marshalling pointers is sometimes pretty simple: one simply transmits the target of the pointer, rather than the pointer. Unmarshalling depends on whether the receiver is a callee (i.e., the pointer is an input parameter) or is a caller (the pointer is an output parameter). For a callee, the pointed-to item is copied into storage on the receiver’s stack in the server-side stub’s frame; the stub passes a pointer to the item to the remote procedure. For a caller, the marshaled item is copied into the original target of the pointer; the pointer itself doesn’t change.
Three situations can complicate the marshalling of pointers. The first is when a pointer contains a null value: since it’s pointing at nothing, there’s nothing to send! The second is when two different pointers point to the same location (this is known as aliasing). It’s not enough to send the value of what the pointer points to: the reconstructed pointers on the receiver must also point to the same location. Lastly, what if the pointer points to a data structure containing another pointer?
Marshalling unrestricted pointers, i.e., pointers that might be null, might be aliased, or might point to data structures containing other pointers, requires that one send a representation of how the data structure is organized. One such representation is illustrated in the slide: what the pointers point to is represented as an array and the pointers are represented as indices of the array.
Referring to Server State

Client

pointer

Server
Rather than pass a tree back and forth between client and server, it might make more sense to leave the tree on the server and have the client merely send the server requests to perform operations on it. The interface shown here has two procedures — one to create and initialize a tree, and another to add new items to it. From the client's point of view, the tree is represented as an opaque pointer of type `context handle`. This is created implicitly (on both client and server) via the use of the `out` parameter of the `create` procedure. The client holds onto this handle; whenever it uses the handle with the `insert` procedure, it is converted on the server side to point to whatever the pointer pointed to that the server originally returned via the `out` parameter of the `create` procedure.

If the server crashes, then, from the client's point of view, the context handle becomes useless. The client will be notified of a server failure if it tries to use the context handle after the server is known to have crashed.

If the client crashes, then the server might want to be notified, especially if the client is the only one interested in the tree represented by the context handle. In the event of a crash, the server runtime will clean up its state. If the server is interested, it can define a cleanup (or `rundown`) procedure. The name of the cleanup procedure in this case would be `tree_t_rundown`, which will be called with the server-side pointer (to the tree) as the argument. The server, in this example, would free the storage that had been allocated for the tree.
It’s often useful to be able to come up with an identifier that’s guaranteed to be unique, even though all computers on the planet are using the same algorithm. The example here is from DCE’s interface definition language. To identify a procedure (or, more precisely, an interface), each interface is assigned a unique number called a UUID — *Universal Unique Identifier*. We discuss UUIDs in more detail on the next page. In addition, interfaces are assigned *version numbers*. These consist of a major and a minor portion. (In the picture, the version number is 3.1, with 3 being the major portion and 1 the minor portion.) Backwards compatibility is provided to callers of the same major, but lower minor number. If the caller and callee have different major version numbers, then they are incompatible.
UUIDs are guaranteed-to-be-unique numbers: the claim is that they are unique for “all” time (at least for the lifetime of the companies using them) in the known universe. This is a rather tall order. It certainly requires lots of bits.

The basic idea is to construct a UUID from the current time of day and the unique ID of the local computer. 60 bits of the UUID are used to hold the time of the creation of the UUID, measured in hundreds of nanoseconds since either October 15, 1582 (the date our current Gregorian calendar went into effect in Catholic Europe: see http://serendipity.magnet.ch/hermetic/cal_stud/cal_art.htm), January 1, 1970 (the beginning of time on Unix systems), or January 1 1980 (the beginning of time on Windows systems). Which one it is depends on the “reserved” field. If UUIDs must be created faster than one every hundred nanoseconds, then a few timestamps can be “borrowed” from the future. However if too many rapid requests for UUIDs are encountered, the generator idles (by looping) until the clock “ticks,” i.e. the current time is greater than that indicated by any of the borrowed timestamps.

A well behaved clock will never move backwards — this is particularly important for guaranteeing the uniqueness of UUIDs. However, the UUID generator keeps track of the last UUID generated. If it detects that the clock has moved backwards, it adjusts for this with the clock sequence number field, which is modified each time the clock is found to have moved backwards.

Forty-eight bits of the UUID are reserved for the address of the machine, which is given in a format known as IEEE 802. As far as anyone can tell, 48 bits is plenty for this.

Windows systems use something similar to UUIDs, but they’re called GUIDs (globally unique identifiers).