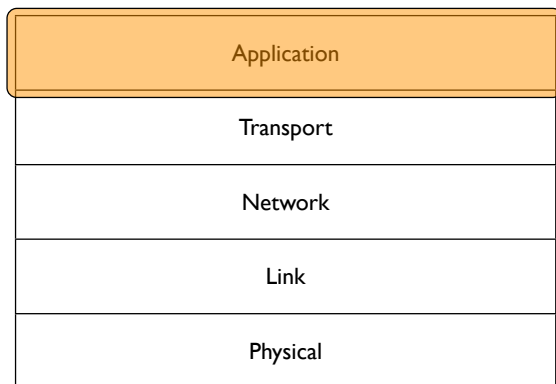
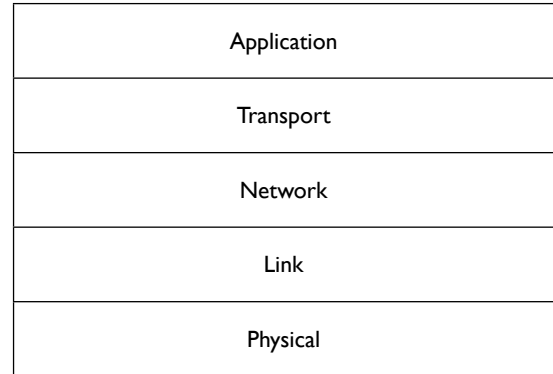


RPC and serialization

Sections 5.3 and 7.1 in the textbook



RPC vs RPC

- RPC can mean two different things, depending on context
- SunRPC (now ONC RPC) was a technology from Sun with its NFS suite
 - Allowed procedures/functions to be called remotely
- Remote procedure call in general can refer to other similar technologies (CORBA, SOAP)

```
int num_records(void);
int process_record(int);
static
int
process_records(void)
{
    int i, n = num_records();
    for (i = 0; i < n; i++)
        if (!process_record(i))
            return 0;
    return 1;
}

struct {
    ...
} *records;
int nrecords;

int
num_records(void)
{
    return nrecords;
}
...

```



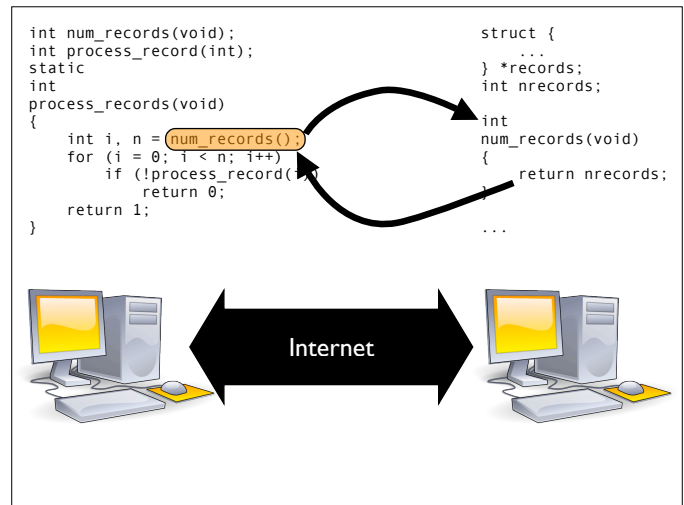
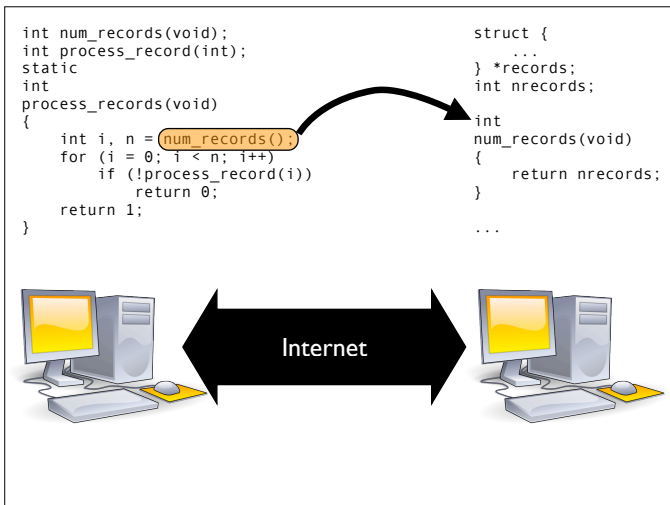
```
int num_records(void);
int process_record(int);
static
int
process_records(void)
{
    int i, n = num_records();
    for (i = 0; i < n; i++)
        if (!process_record(i))
            return 0;
    return 1;
}

struct {
    ...
} *records;
int nrecords;

int
num_records(void)
{
    return nrecords;
}
...

```





Programming language matters

- Before looking at the networking issues...
 - How can we make this transparent to C?
- Each function definition becomes three function definitions:
 - Function definition
 - Stub
 - Skeleton

Stubs and skeletons

- “Client” refers to the party making the function call
- “Server” refers to the party implementing the function
- On the client side, when we call `num_records()`, we’re actually calling a stub

```

int
num_records(void)
{
    ... generate an RPC request packet ...
    write(socket, rpc_packet, sizeof rpc_packet);
    read(response, rpc_packet, rpc_packet);
    ... figure out if it succeeded, etc., put
    returned value into “ret” ...
    return ret;
}

```

```

void
num_records_skeleton(struct svc_req *rfc_packet)
{
    ... verify rfc_packet is valid ...
    int r = num_records();
    ... construct rfc return packet ...
    write(socket, rfc_return_packet, ...);
}

```

Not quite

- The last few slides were a bit of a lie
- SunRPC doesn’t do stubs and skeletons automatically for you; it exposes an API to you
- Most modern RPCs (not compatibly with SunRPC) automatically generate stubs and skeletons so it’s all transparent

Distributed Objects Programming Topics

Introduction
Articles
About Distributed Objects
Distributed Objects Architecture
Connections and Proxies
Ports and Name Servers
Message Encapsulation
Vending an Object
Getting a Vended Object
Configuring a Connection
Handling Connection Errors
Authenticating Connections
Making Substitutions During Message Encoding
Using NSInvocation
Revision History

Related Reference
Objective-C
NSConnection

Developer Connection

Log In | Not a Member? | Contact ADC

ADC Home > Reference Library > Guides > Cocoa > Interapplication Communication > Distributed Objects Programming Topics >

Hide TOC | Next Page >

Introduction to Distributed Objects

Contents:
Limitations
Organization of This Document

The Objective-C runtime supports an interprocess messaging solution called "distributed objects." This mechanism enables a Cocoa application to call an object in a different Cocoa application (or a different thread in the same application). The applications can even be running on different computers on a network.

This programming topic describes the Cocoa classes that form the distributed objects system.

Limitations

eXternal Data Representation

- SunRPC does not define a networking protocol
- I have yet to come across an RPC system which defines its own networking protocol directly
- RPC layers sit atop serialization layers

XDR

- This was Sun's equivalent (before Sun created Java, ha ha) to Java's implements Serializable
- Abstracts over all the serialization issues
- Requires an extra compilation step:
 - rpcgen compiles Interface Description Files down to C code
 - C compiler generates executables

```
const MAXUSERNAME = 32; /* max length of a user name */
const MAXFILELEN = 65535; /* max length of a file */
const MAXNAMELEN = 255; /* max length of a file name */

/*
 * Types of files:
 */
enum filekind {
    TEXT = 0, /* ascii data */
    DATA = 1, /* raw data */
    EXEC = 2 /* executable */
};

/*
 * File information, per kind of file:
 */
union filetype switch (filekind kind) {
    case TEXT:
        void; /* no extra information */
    case DATA:
        string creator<MAXNAMELEN>; /* data creator */
    case EXEC:
        string interpreter<MAXNAMELEN>; /* program interpreter */
};

/*
 * A complete file:
 */
struct file {
    string filename<MAXNAMELEN>; /* name of file */
    filetype type; /* info about file */
    string owner<MAXUSERNAME>; /* owner of file */
    opaque data<MAXFILELEN>; /* file data */
};
```

5.2.4.2.1 Sizes of integer types <limits.h>

The values given below shall be replaced by constant expressions suitable for use in #if preprocessing directives. Moreover, except for CHAR_BIT and MB_LEN_MAX, the following shall be replaced by expressions that have the same type as would an expression that is an object of the corresponding type converted according to the integer promotions. Their implementation-defined values shall be equal or greater in magnitude (absolute value) to those shown, with the same sign.

14) See "Annoying language directions" (6.11.3).

5.2.4.2.1 Environment 21

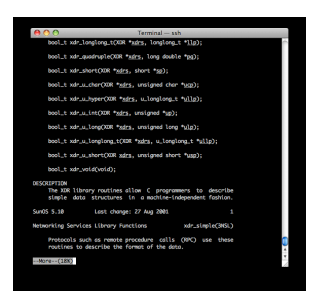
INTEGR 9099:1999 (E)	CISO/IEC
— number of bits for smallest object that is not a bit field (byte)	CHAR_BIT 8
— minimum value for an object of type signed char	SCHAR_MIN -127 // -(2 ⁷ - 1)
— maximum value for an object of type signed char	SCHAR_MAX 127 // 2 ⁷ - 1
— maximum value for an object of type unsigned char	UCHAR_MAX 255 // 2 ⁸ - 1
— minimum value for an object of type char	CHAR_MIN see below
— maximum value for an object of type char	CHAR_MAX see below
— maximum number of bytes in a multibyte character, for any supported locale	MB_LEN_MAX 1
— minimum value for an object of type short int	SHRT_MIN -32767 // -(2 ¹⁵ - 1)
— maximum value for an object of type short int	SHRT_MAX 32767 // 2 ¹⁵ - 1
— maximum value for an object of type unsigned short int	USHRT_MAX 65535 // 2 ¹⁶ - 1
— minimum value for an object of type int	INT_MIN -32767 // -(2 ¹⁵ - 1)
— maximum value for an object of type int	INT_MAX 32767 // 2 ¹⁵ - 1
— maximum value for an object of type unsigned int	UINT_MAX 65535 // 2 ¹⁶ - 1
— minimum value for an object of type long int	LONG_MIN -2147483647 // -(2 ³¹ - 1)
— maximum value for an object of type long int	LONG_MAX 2147483647 // 2 ³¹ - 1
— minimum value for an object of type unsigned long int	ULONG_MAX 4294967295 // 2 ³² - 1
— minimum value for an object of type long long int	LLONG_MIN -9223372036854775807 // -(2 ⁶³ - 1)
	LLONG_MAX 9223372036854775807 // (2 ⁶³ - 1)

22 Environment 5.2.4.2.1

- No exact sizes
- No guarantees of endianness
- No representation of signedness
- No representation of floating point numbers

XDR assumptions

- Converts from native representations to:
 - 8-bit bytes
 - 2's complement
 - IEEE floats
 - 32-bit longs
 - et cetera



Character encoding

- Sadly we don't have time to do this properly :(
- ASCII is just the tip of the iceberg
 - 7-bit code, only 95 printable characters
- Unicode is a 32-bit character space but it is not a representation

Unicode representation

- UCS-4: each character is 4 bytes
- UCS-2: truncated UCS-4
- Code pages
 - Map in 128 characters from somewhere in the 32-bit space
- 95 + 128 = 223 printable characters
- UTF-8 and UTF-16
 - Variable-length codes
 - UTF-16 deprecates UCS-2 and is very complex
 - UTF-8 popular!

UTF-8

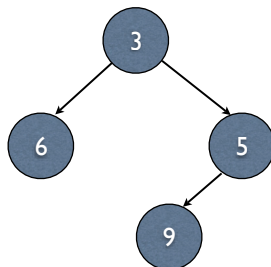
- Remember Huffman coding
 - That's great, but UTF-8 isn't Huffman coding
 - But it does accomplish almost the same thing and is very easy to encode/decode (no table/tree required)!
- ASCII values can be represented in 1 byte

UTF-8

U+000000–U+00007F	0xxxxxxx	ASCII ("Basic Latin")
U+000080–U+0007FF	110yyyyx 10xxxxxx	Latin extended, combining, Greek, Cyrillic, Hebrew, etc.
U+000800–U+00FFFFF	1110yyyy 10yyyyxx 10xxxxxx	All Human writing, math symbols, dingbats, etc.
U+010000–U+10FFFF	11110zzz 10zzyyyy 10yyyyxx 10xxxxxx	Ancient Greek, Phoenician, Klingon, etc.
...	...	???

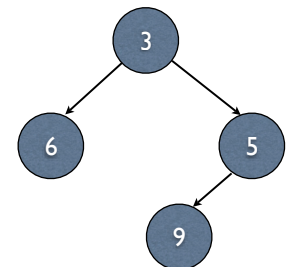
What about pointers?

- We can't just ship off pointer values
- Idea #1: we allow the server to make requests back to the client asking to look up certain memory locations
- Idea #2: during serialization we chase down all the pointers



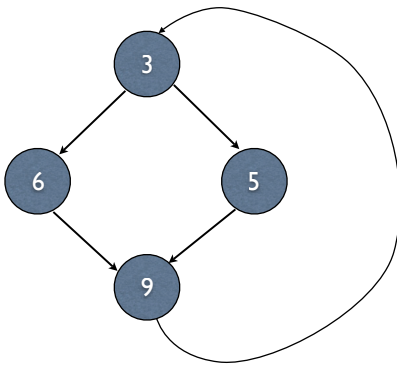
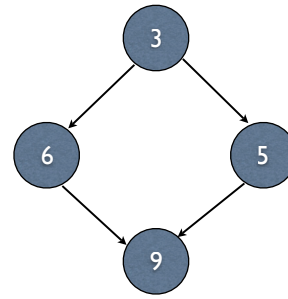
Chasing down pointers

- Take a pointer-y object and "flatten" it down
- E.g., <3, <6, <>, <>>, <5, <9, <>, <>>, <>>>
- becomes <1, 3, 1, 6, 0, 0, 1, 5, 1, 9, 0, 0, 0>



Pointers and serialization

- XDR (and most systems) typically go for the “chasing down pointers” route
- XDR allows you (and helps you) to define procedures to automatically chase down pointers



```
struct rpc_msg {
    unsigned int xid;
    union switch (msg_type mtype) {
        case CALL:
            call_body cbody;
        case REPLY:
            reply_body rbody;
    } body;
};

struct call_body {
    unsigned int rpcvers; /* must be equal to two (2) */
    unsigned int prog;
    unsigned int vers;
    unsigned int proc;
    opaque_auth cred;
    opaque_auth verf;
    /* procedure specific parameters start here */
};

Body of a reply to an RPC call:

union reply_body switch (reply_stat stat) {
    case MSG_ACCEPTED:
        accepted_reply areply;
    case MSG_DENIED:
        rejected_reply rreply;
} reply;
```

```
struct rpc_msg {
    unsigned int xid;
    union switch (msg_type mtype) {
        case CALL:
            call_body cbody;
        case REPLY:
            reply_body rbody;
    } body;
};

struct call_body {
    unsigned int rpcvers; /* must be equal to two (2) */
    unsigned int prog;
    unsigned int vers;
    unsigned int proc;
    opaque_auth cred;
    opaque_auth verf;
    /* procedure specific parameters start here */
};

Body of a reply to an RPC call:

union reply_body switch (reply_stat stat) {
    case MSG_ACCEPTED:
        accepted_reply areply;
    case MSG_DENIED:
        rejected_reply rreply;
} reply;
```

RPC fields

- Transaction ID: random/arbitrary number used to distinguish multiple requests
- Program ID: used by the RPC server to know which program implements the function (e.g., 0x00100003 = NFS)
- Function ID: because we don't want to have to send function names as strings

Procedure call semantics

- Local procedure calls (not using RPC) implement at-most-once semantics
- You call a function. That function will return 0 (infinite loop? crash?) or 1 times
- This is a nice property to have in RPC, though SunRPC doesn't since it uses UDP
 - Why can't we get at-most-once?

Procedure call semantics

- Local procedure calls are synchronous
 - The caller cannot continue execution until the callee returns
- RPC is typically synchronous, but asynchronous is common as well
 - The caller will be alerted when/if the reply arrives