Client / Server Model using Message Passing (Review)

- **Client / server model**
  - **Server** = process (or collection of processes) that provides a service
    - Example: name service, file service
  - **Client** — process that uses the service
  - Request / reply protocol:
    - Client sends **request** message to server, asking it to perform some service
    - Server performs service, sends **reply** message containing results or error code

Why is Message Passing not Ideal?

- Disadvantages of client-server communication via message passing:
  - Message passing is I/O oriented, rather than request/result oriented
  - Programmer has to explicitly code all synchronization
  - Programmer may have to code format conversion, flow control, and error control

- **Goal — heterogeneity** — support different machines, different OSs
  - Portability — applications should be trivially portable to machines of other vendors
  - Interoperability — clients will always get same service, regardless of how vendor has implemented that service
  - OS should handle data conversion between different types of machines

Remote Procedure Call (RPC)

- **RPC mechanism:**
  - Hides message-passing I/O from the programmer
  - Looks (almost) like a procedure call — but client invokes a procedure on a server

- **RPC invocation (high-level view):**
  - Calling process (client) is suspended
  - Parameters of procedure are passed across network to called process (server)
  - Server executes procedure
  - Return parameters are sent back across network
  - Calling process resumes

- Invented by Birrell & Nelson at Xerox PARC, described in February 1984 *ACM Transactions on Computer Systems*

RPC Invocation

- Each RPC invocation by a client process calls a **client stub**, which builds a message and sends it to a **server stub**
- The server stub uses the message to generate a local procedure call to the server
- If the local procedure call returns a value, the server stub builds a message and sends it to the client stub, which receives it and returns the result(s) to the client
Parameter Passing

- **Parameter marshaling** — client stub packs parameters into a message

```
nr_hits = query(key, 10, result);

marshal
```

- **Parameter unmarshaling** — server stub unpacks parameters for local procedure

```
rn_hits = query(key, 10, result);

unmarshal
```

Parameter Passing (cont.)

- Handle different internal representations
  - ASCII vs. EBCDIC vs. …
  - 1’s comp. vs. 2’s comp. vs. floating-point
  - Little endian vs. big endian
  - Establish a canonical (standard) form?

- What types of passing are supported?
  - Remote procedure can’t access global variables — must pass all necessary data
  - **Call-by-value** (procedure gets a copy of data) — pass parameters in message
  - **Call-by-reference** (procedure gets a pointer to data)
    - Can’t do call-by-reference
    - Do call-by-copy / restore instead
      - Instead of pointer, pass item pointed to
      - Procedure modifies it, then pass it back
    - Inconsistency if client doesn’t block

Generating Stubs

- C and C++ may not be descriptive enough to allow stubs to be generated automatically

```
typedef struct {
  double item1;
  int item2;
  char *annotation;
} tuple;
```

- Which are in, in-out, and out parameters?
- Exactly what size are parameters (e.g., integers, arrays)?
- What does it mean to pass a pointer?

- Using OSF’s DCE Interface Definition Language (IDL) to specify procedure signatures for stub generation:

```
interface db
{
typedef struct {
  [in] long key,
  [in] tuple value
} tuple;

boolean add {
  [in] long key,
  [in] tuple value
};

boolean remove {
  [in] long key,
  [in] tuple value
};

char add(int key, tuple value);
char remove(int key, tuple value);
int query(int key, int number, tuple values[]);
}
```

Processes vs. Threads

- **Process** = unit of resource ownership
  - A process (sometimes called a heavy weight process) has:
    - Address space
    - Program code
    - Global variables, heap, stack
    - OS resources (files, I/O devices, etc.)

- **Thread** = unit of scheduling
  - A thread (sometimes called a lightweight process) is a single sequential execution stream within a process
  - A thread shares with other threads:
    - Address space, program code
    - Global variables, heap
    - OS resources (files, I/O devices)
  - A thread has its own:
    - Registers, Program Counter (PC)
    - Stack, Stack Pointer (SP)
Processes vs. Threads

- A thread is bound to a particular process
  - A process may contain multiple threads of control inside it
  - Threads can block, create children, etc.
- All of the threads in a process:
  - Share address space, program code, global variables, heap, and OS resources
  - Execute concurrently (has its own register, PC, SP, etc. values)

Using Threads in a Server

- Dispatcher-worker model
  - Dispatcher thread receives all requests, hands each to an idle worker thread, worker thread processes request
  - Worker threads are either created dynamically, or a fixed-size pool of workers is created when the server starts
- Team model
  - All threads are equals; each thread processes incoming requests on its own
  - Good for handling multiple types of requests within a single server
- Pipeline model
  - First thread partially processes request, then hands it off to second thread, which processes some more, then hands it off to third thread, etc.

User-Level vs. Kernel-Level Threads

- User-level threads = provide a library of functions to allow user processes to create and manage their own threads
  - ✔ No interaction with OS (= fast)
  - ✔ User can customize CPU scheduling
  - ✘ Process gets one time slice
  - ✘ If one thread blocks whole process blocks
- Kernel-level threads = kernel provides system calls to create and manage threads
  - ✔ Kernels knows about all threads
  - ✔ Can give process larger time slice
  - ✔ Can switch to other thread if one blocks
  - ✘ Interaction with OS (= slow)
- Most modern OS’s support both

Two-Level Thread Model
(Digital UNIX, Solaris, IRIX, HP-UX)

- User-level threads for user processes
  - “Lightweight process” (LWP) serves as a “virtual CPU” where user threads can run
- Kernel-level threads for use by kernel
  - One for each LWP
  - Others perform tasks not related to LWPs
- OS supports multiprocessor systems