Distributed Systems 1

CUCS Course 4113
https://systems.cs.columbia.edu/ds1-class/

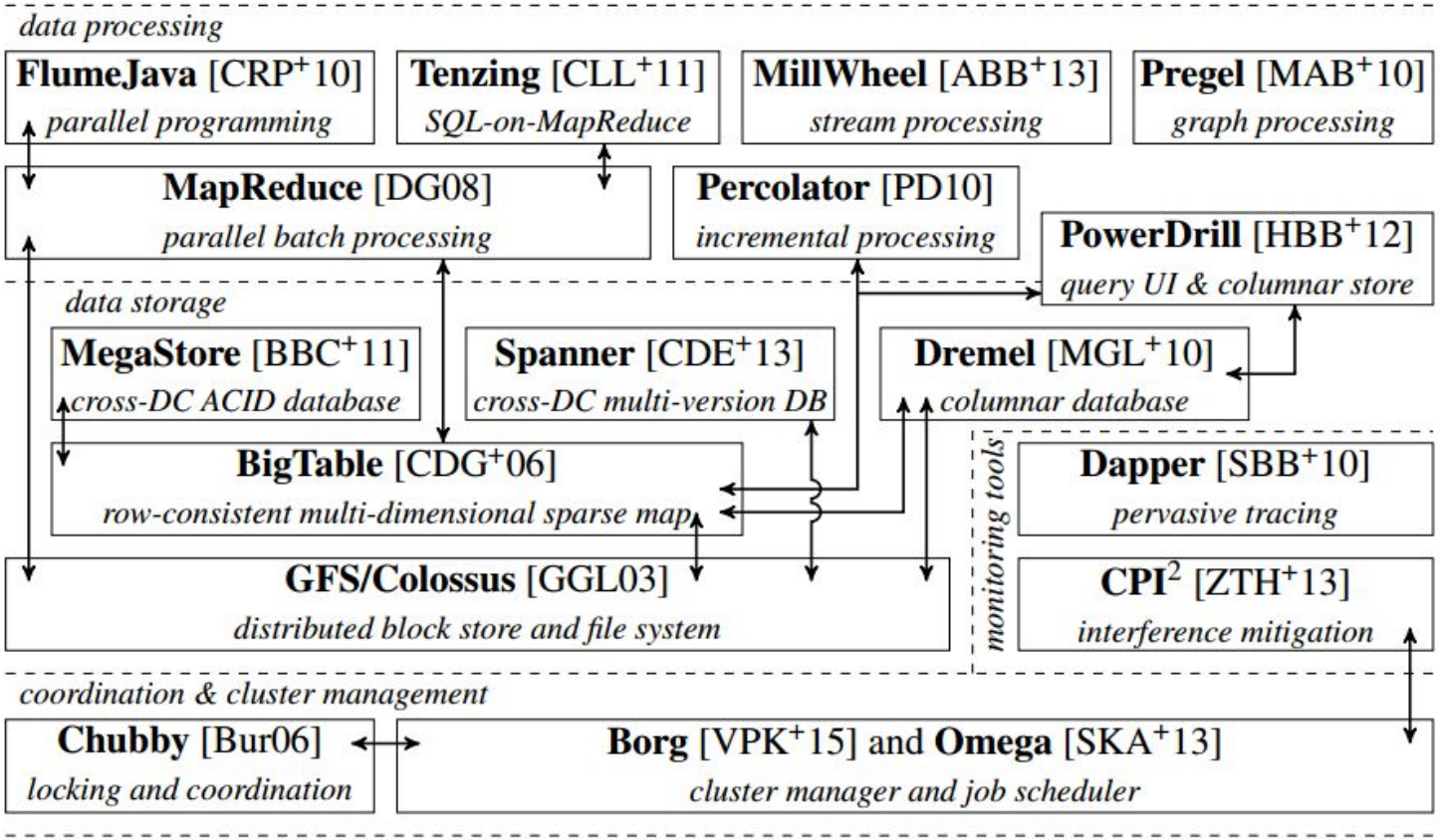
Instructor: Roxana Geambasu

Remote Procedure Calls (RPC)

Context

- We discussed challenges of achieving scalability, fault tolerance, semantics, performance in DSes.
- The approach is generally to construct infrastructure systems that raise the level of abstraction.
- Today we look at the most basic DS abstraction: **RPC**, the predominant communication abstraction in a DS.
- RPC already reflects most challenges we talked about.

Example: (Part of) Google Infra Stack



Motivation

- To coordinate, nodes must communicate.
- Network gives sockets, which are terrible to program with.
- What are some issues?

Example (code sketch!)

```
struct foomsg {
   u_int32_t len;
}
send_foo(int outsock, char* contents) {
   int msglen = sizeof(struct foomsg) +
                    strlen(contents);
   char* buf = malloc(msglen);
   struct foomsg* fm = (struct foomsg*)buf;
   fm->len = htonl(strlen(contents));
   memcpy(buf + sizeof(struct foomsg),
              contents,
              strlen(contents));
   write(outsock, buf, msglen);
}
```

Motivation

- To coordinate, nodes must communicate.
- Network gives sockets, which are terrible to program with.
- What are some issues?
  - Lots of ugly boilerplate
  - Prone to bugs, vulnerabilities
  - Portability issues
  - Hard to understand/maintain/evolve

Example (code sketch!)

```
struct foomsg {
   u_int32_t len;
}
send_foo(int outsock, char* contents) {
   int msglen = sizeof(struct foomsg) +
                    strlen(contents);
   char* buf = malloc(msglen);
   struct foomsg* fm = (struct foomsg*)buf;
   fm->len = htonl(strlen(contents));
   memcpy(buf + sizeof(struct foomsg),
              contents,
              strlen(contents));
   write(outsock, buf, msglen);
}
```

RPC

(~1984 paper by Birrell, Nelson)

- Idea: Make network communication look like a *local procedure call* (LPC).

Caller (Client)

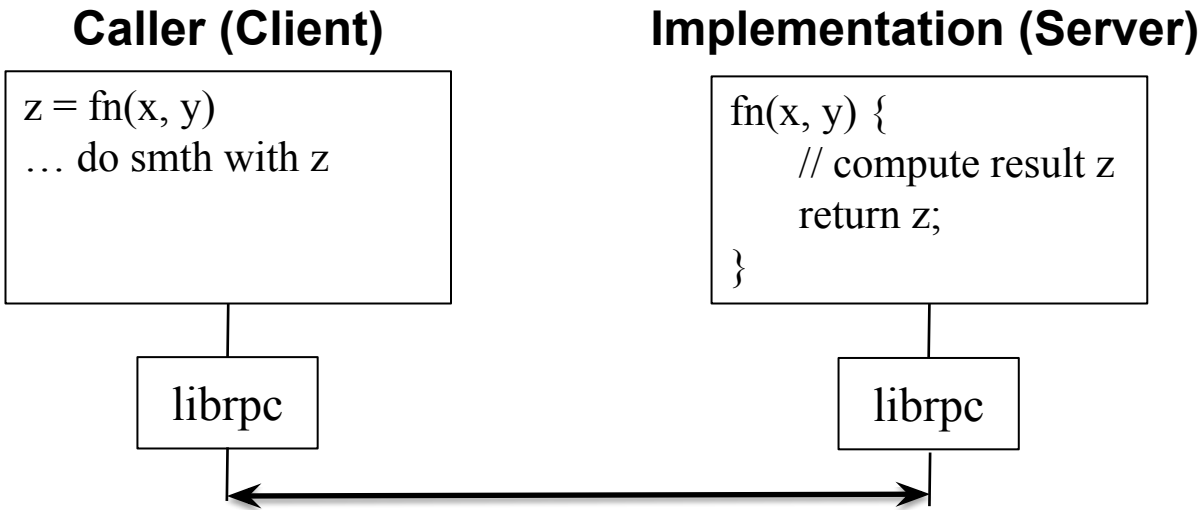
```
z = fn(x, y)
… do smth with z
```

Implementation (Server)

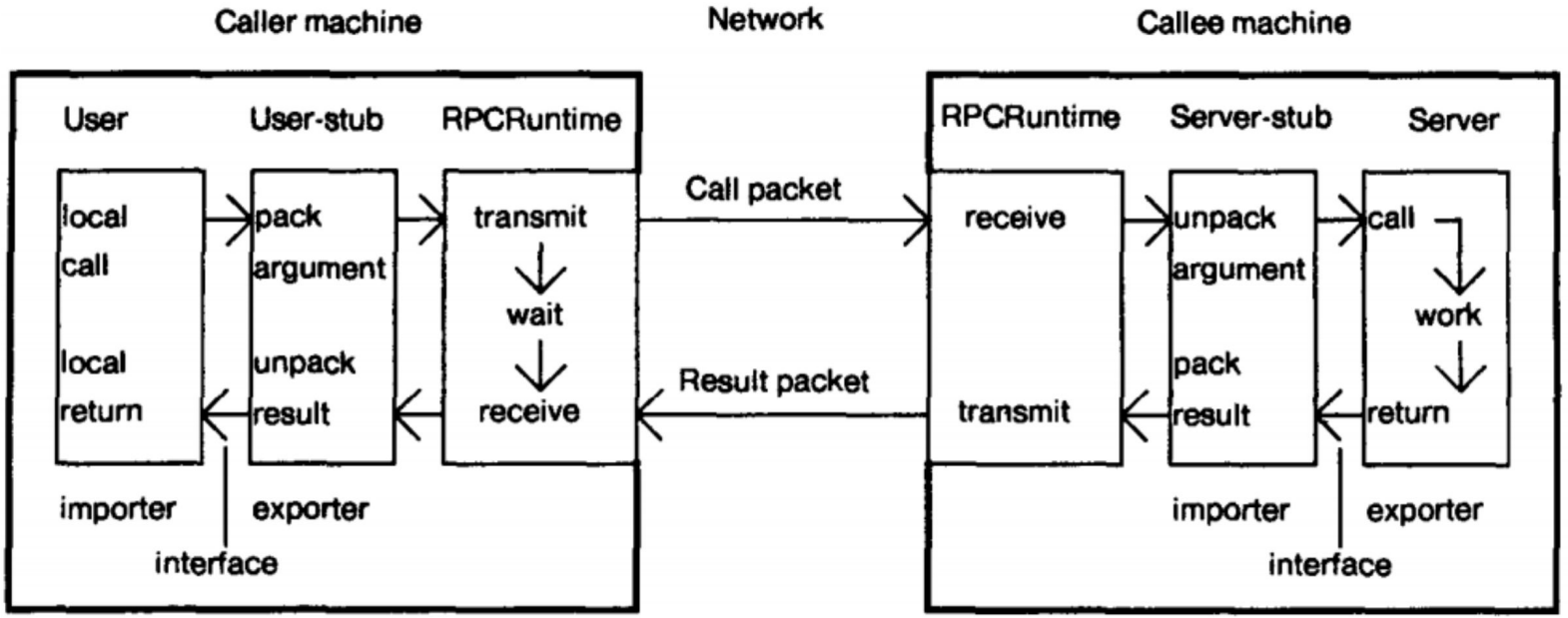
```
fn(x, y) {
      // compute result z
      return z;
}
```

RPC

- Idea: Make network communication look like a *local procedure call* (LPC).



RPC Architecture



(figure taken from RPC paper by Birrell, Nelson)

Benefits

- Easy to use and familiar to any programmer.
- Hides gory network/marshaling details that one would have to implement if doing, e.g., network-level communication, byte orders, …
- Supports evolution of the communicating components independently.
- Allows for efficient packaging of arguments/return vals.
- Authentication support.
- Location independence.

Problems

(or where distribution peeks through the LPC illusion)

Problems

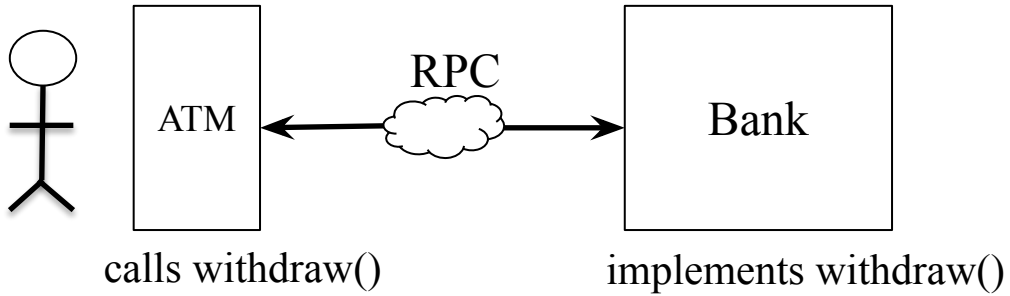
(or where distribution peeks through the LPC illusion)

- Latency
  - LPC: fast; RPC: can be slow.
  - So care must be taken when invoking RPCs.
- Pointer transfers
  - LPC: caller/callee share address space; RPC: no shared mem.
  - RPClib can't automatically decide what gets serialized and what doesn't.
- Failures
  - LPC: shared fate between caller and callee. RPC: caller and callee can **fail independently** (recall DS definition).
  - This is the critical challenge in DS and why cannot hide distribution.

Example: ATM

BREAKOUT

- **Breakout activity (5 min):** Design an ATM.
  - When a person wants to withdraw cash from the ATM, the ATM sends an RPC to the bank. The bank first checks that the person has enough money in their account, and if so, deducts the money and confirms with the ATM. The ATM, in turn, should give the money to the user.



RPC Semantics

- At least once
- At most once
- Exactly once

At Least Once

- Semantic: RPC is eventually executed at least once, but potentially multiple times.
- Implementation:
  - Client keeps issuing RPC until gets a response from server.
  - If failures (of net/server) are temporary, semantic satisfied.
- Problem: Suitable for some but not other functions.
  - Generally, suitable for *idempotent operations* (issued once vs. multiple times doesn't change state).

At Most Once

- Semantic: RPC is executed zero or one times, not more.
- Implementation:
  - Clients identify their requests (xid).
  - Server remembers xids to detect duplicates and squelch them.
- Problem: server failure at inopportune time can cause failure of the semantic. Give examples.

```
if (state[xid] == DONE)
    return r[xid];
x = handler(args)
state[xid] = DONE
response[xid] = x
return x
```

Exactly Once

- Semantic: RPC is executed once.
- This is the ideal (it resembles the LPC model most closely and it's easiest to understand), but it's surprisingly hard to implement.
- The clean solution is to implement the handler() in a *transaction* together with state[xid], response[xid] operations.

Take-Aways

- Even this most basic abstraction in a DS, RPC, requires a lot of complexity to implement with strong semantics.
- *Strong semantic here* means equivalent functionality to that of the non-DS version of the program.
- Independent failures by different components of a DS cause these complications through DS stack.

RPC Libs

- Your HW series includes a basic RPC lib in the skeleton.
- For your other projects/work, you should consider using a publicly available, popular library.
- We give a few examples here of RPC libs, but there are many more out there that are worth considering: https://columbia.github.io/ds1-class/lectures/03-rpc-handout.pdf.