Client-Server Model

Paulo Sérgio Almeida

Grupo de Sistemas Distribuídos
Departamento de Informática
Universidade do Minho

Client-server paradigm

- Classic paradigm in distributed systems
- Very general and independent from tools/middleware used:
  - sockets,
  - ONC RPC,
  - SOAP . . .
- Basic block in building sistems and describing paradigms:
  - N-tier: client and server chaining;
  - callbacks: “clients” are also servers;
  - distributed objects: servers implement objects.

Client and server as roles

- Client-server model is asymmetric;
- Server: passive process which is contacted by clients; when contacted, processes request and sends reply.
- Client: contacts server towards using a service; sends request and waits for reply.
- Client and server are roles to be played.
- Any entity can be both server and client.

  To serve a request, a server may need to use another service, becoming client to the latter.

Request-reply protocol

- A protocol which, using a transport protocol (e.g., TCP or UDP), allows conversation between client and server.
- Some needs:
  - delimit requests;
  - encode which operation is requested;
  - delimit arguments to operation.
- Sharing transport medium may lead to:
  - encode client thread/process id;
  - encode server entity id;
- Coping with failures may lead to:
  - encode request id (e.g., client and sequence number);

Data structure serialization

- Clients and servers operate on local data structures;
- Making a request involves serialising data structures;
- Hardware heretoregeneity leads to neutral serialization formats.

  Impossible to contemplate all hardware combinations.

- Formats can be binary (e.g., XDR, CDR).

  These are compact and allow a CPU efficient serialization.

- Or can be text based:

  With increasing popularity post Web; costly due to parsing and the native-text and text-native conversions.

Request dispatching

A server can adopt different models to serve requests:

- Sequential server
- Process per client
- Thread per client
- Thread per request
- Thread pool

Sequencial server

- A single process handles each client in sequence.
- Only viable for quickly handled clients/requests.
- Does not scale with number of clients/requests, even having lots of processing power:

  Network latency.

- Vulnerable to slow or malicious clients.

  Denial of service.

Process per client

- Classic server model in UNIX: `fork` per `accept`.
- Easy to program and robust: avoids nasty concurrency control problems.
- Difficult to maintain shared state: inter-process communication.
- Appropriate under little interaction between clients (through server).

Thread per client

- Allows maintaining shared state in memory.
- Requires care with concurrency control in the server.

  Subtle bugs can occur.

- Appropriate for scenarios with dependencies between clients. E.g., blocking operations depending on other clients.

  Producer-consumer workflow.

- Does not scale to a large number of simultaneous clients (even if processing power sufficient).

Thread per request

- Each request is handled in separate thread.
- Extra cost of thread criation.
- Allows more potential for parallelism, but tipically not relevant.
  - the major source of parallelism comes from serving different clients;
  - each client is tipically sequential.
- Allows dependencies between requests coming from the same multi-threaded client. (Relevant under connection sharing.)

  Maximizes location transparency.

Thread pool

- Uses a pre-created pool of threads to serve requests.
- May assign thread per client or per request.
- Intends to save costs of thread creation and recicling.
- Fixed pool plus blocking operations may lead to deadlock.
- Pool may be dynamically adjusted according to pending operations.
- Less important nowadays with OS support for efficient thread creation (e.g., in Linux).

Connection management

Different alternatives for using connections:

- Connectionless (e.g., using UDP): appropriate for small sized requests and custom error handling.
- Connection per request: inneficient, due to connection establishment cost. Used in first version of HTTP.
- Connection per client: a client establishes a connection, which is used for all requests.

And the multi-threaded clients?

- a multi-threaded client can be seen as several independent clients; leads to a waste of connections.
- Solution: share a single connection for different client threads.
  - Complicates implementation.
  - Interaction with dispatching model.
  - Need to relate requests to client threads.
  - Atomic versus blocking operations.

Servers without conversational state: stateless servers

- Stateless server:
  - each request contains all information needed to be handled;
  - server replies to a request independently from previous ones;
  - “session” state is kept in client;
- The notion of stateless server is important:
  - without no need to maintain conversational state, less resources are needed, leading to greater scalability and fault tolerance.
  - No need to maintain and recicle per-client state.
  - If a client fails the server does not care.
  - If the server fails it can be restarted and clients resume sending requests.

Example: distributed file systems

- Distributed file systems, like NFS, gain in being implemented with stateless servers.
- Compare:

```
fd = open(fich);
read(fd, buffer, count);
write(fd, buffer, count);
close(fd);
```

- with:

```
read_file_from_to(fich, from, count, buffer);
write_file_from_to(fich, from, count, buffer);
```

- Traditional operations like `open` may be implemented in the client with local invocations to `read_file_from_to` ...
- “Session” state is kept in the client: it is the client application and not the server that has the notion of session.

Idempotency

- Idempotent operations: have the same effect if executed one or more times.
- Example: write some content in a given file block.
- Counter example: add some ammount to a bank account.
- Idempotent operations are more robust:
  - May be used under weaker fault tolerance assurances.
  - If there is a failure (or suspicion of) they can be repeated with no problems.

Obsolescence

- Obsolescence: when an operation becomes redundant given another, e.g., more up-to-date one.
- Example: registering the value of a temperature sensor (when no history is required).
- Obsolescence can be exploited in some situations:
  - avoiding retransmissions under message loss;
  - removing requests from sending or receiving queues avoiding some redundant computation.