Remote Procedure Call

Jingzhu He

Why RPCs

- RPC = Remote Procedure Call
- Proposed by Birrell and Nelson in 1984
- Important abstraction for processes to call functions in other processes
- Allows code reuse
- Implemented and used in most distributed systems, including cloud computing systems
- Counterpart in Object-based settings is called RMI (Remote Method Invocation)
- What's the No. 1 design choice for RPC?

Local Procedure Call (LPC)

- Call from one function to another function within the same process
  - Uses stack to pass arguments and return values
  - Accesses objects via **pointers** (e.g., C) or by **reference** (e.g., Java)

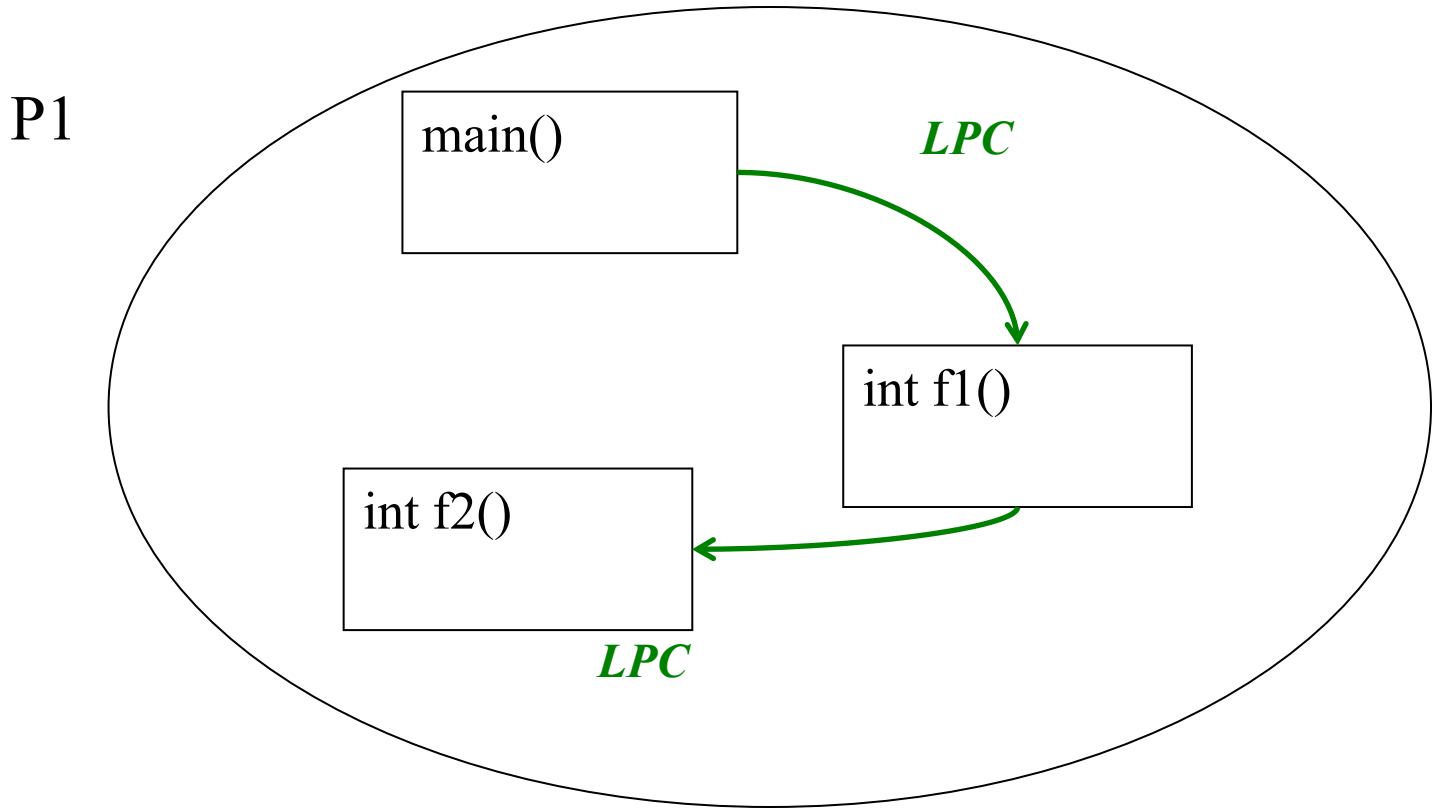
- LPC has **exactly-once** semantics
  - If process is alive, called function executed exactly once

Remote Procedure Call

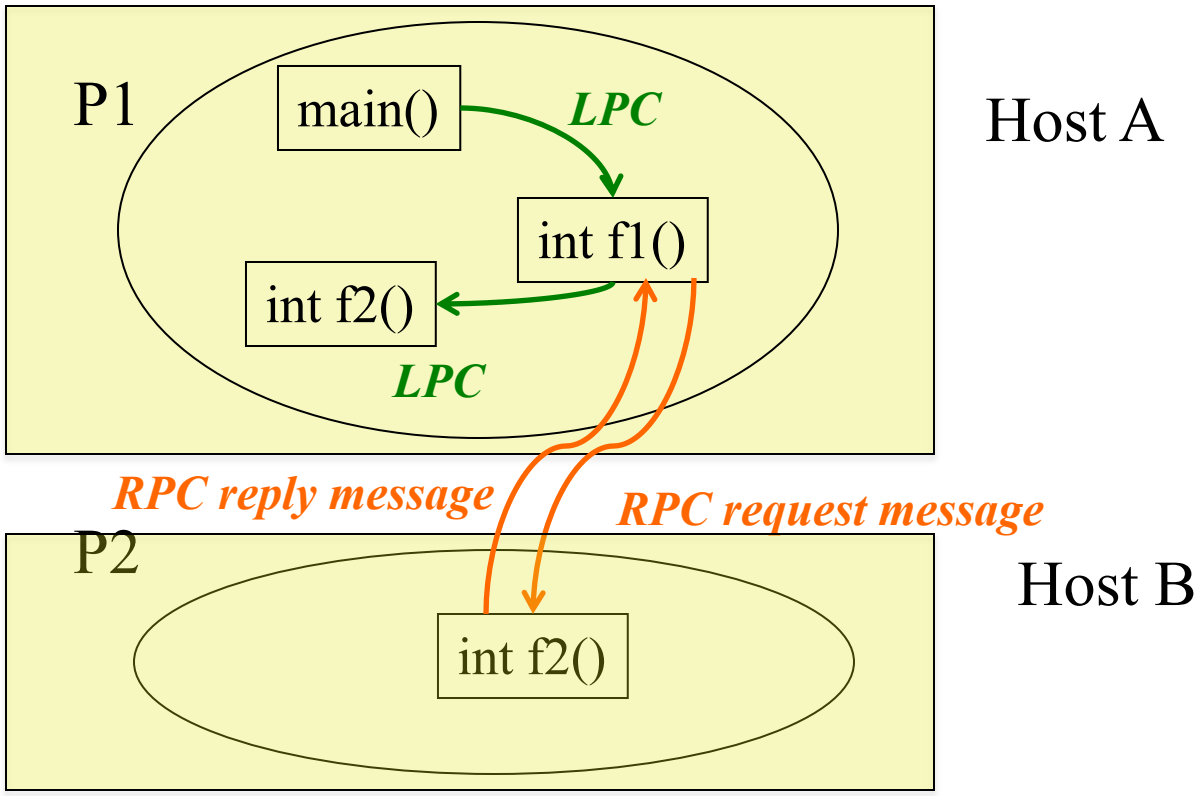
- Call from one function to another function, where caller and callee function reside in different processes
  - Function call crosses a process boundary
  - Accesses procedures via **global references** (e.g., Procedure address = IP + port + procedure number)
  - **Cannot use pointers** across processes since a reference address in process P1 may point to a different object in another process P2

- Similarly, RMI (Remote Method Invocation) in Object-based settings

LPCs



RPCs


P1
main()
LPC
int f1()
int f2()
LPC
Host A
RPC reply message
RPC request message
P2
int f2()
Host B

RPC Call Semantics

- Under failures, hard to guarantee **exactly-once** semantics
- Function may not be executed if
  - Request (call) message is dropped
  - Reply (return) message is dropped
  - Called process fails before executing called function
  - Called process fails after executing called function
  - Hard for caller to distinguish these cases

- Function may be executed multiple times if
  - Request (call) message is duplicated

Implementing RPC Call Semantics

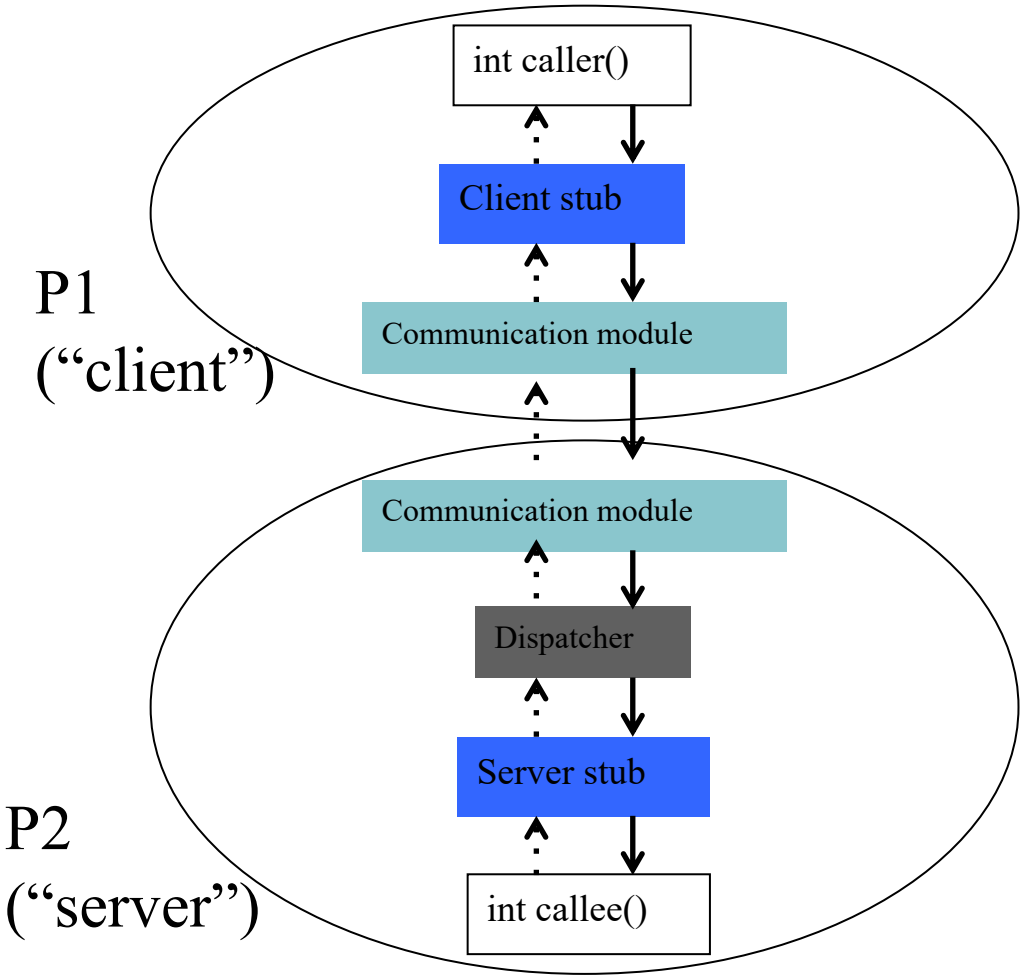
- Possible semantics
  - **At most once** semantics (e.g., Java RMI)
  - **At least once** semantics (e.g., Sun RPC)
  - Maybe, i.e., best-effort (e.g., CORBA)

Retransmit request	Filter duplicate requests	Re-execute function or retransmit reply	RPC Semantics
Yes	No	Re-execute	At least once
Yes	Yes	Retransmit	At most once
No	NA	NA	Maybe

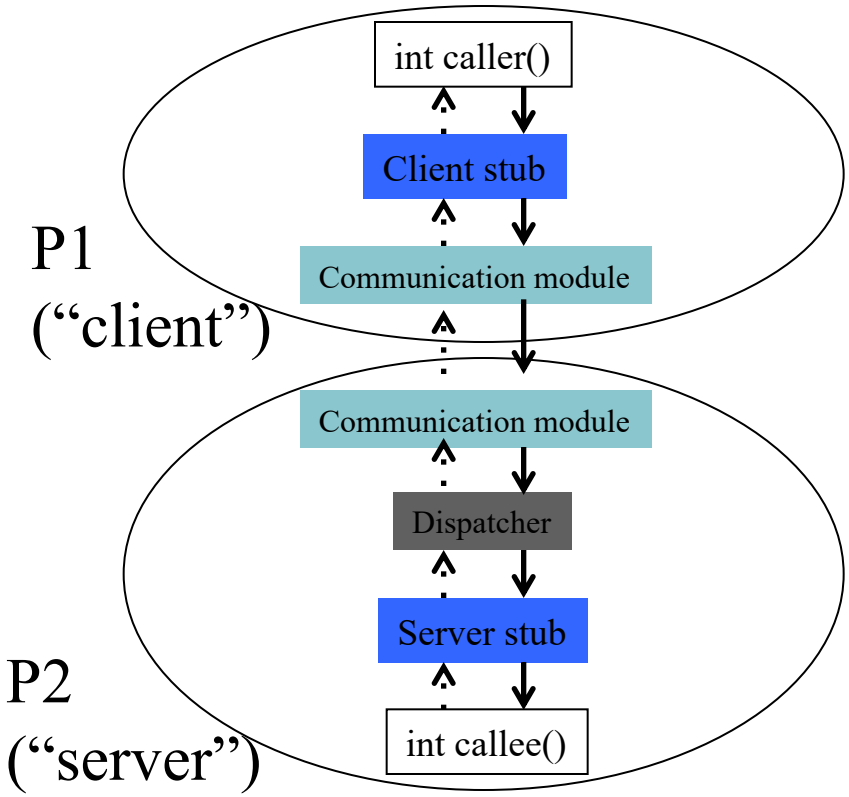
Idempotent Operations

- Idempotent operations are those that can be repeated multiple times, without any side effects
- Examples (x is server-side variable)
  - x=1;
  - x=(argument) y;
- Non-examples
  - x=x+1;
  - x=x*2;
- **Idempotent** operations can be used with **at-least once** semantics

Implementing RPCs



RPC Components



- **Client stub**: has same function signature as callee()
  - Allows same caller() code to be used for LPC and RPC
- **Communication Module**: Forwards requests and replies to appropriate hosts
- **Dispatcher**: Selects which server stub to forward request to
- **Server stub**: calls callee(), allows it to return a value

Generating Code

- Programmer only writes code for **caller** function and **callee** function
- Code for remaining components all **generated automatically** from function signatures (or object interfaces in Object-based languages)
  - E.g., Sun RPC system: Sun XDR interface representation fed into rpcgen compiler
- These components together part of a Middleware system
  - E.g., CORBA (Common Object Request Brokerage Architecture)
  - E.g., Sun RPC
  - E.g., Java RMI

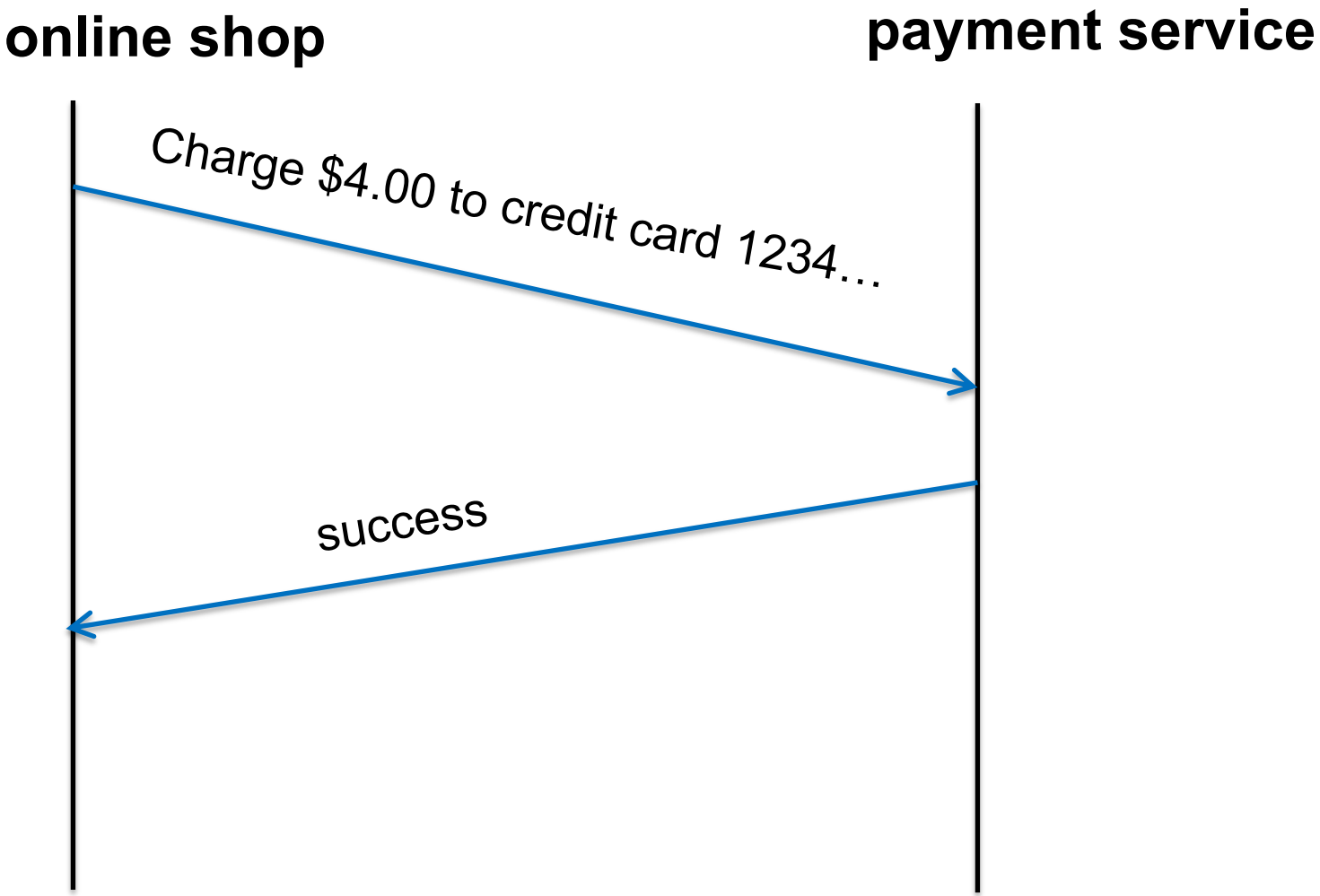
Marshalling

- Middleware has a common data representation (CDR)
  - Platform-independent
- Caller process converts arguments into CDR format
  - Called “Marshalling”
- Callee process extracts arguments from message into its own platform-dependent format
  - Called “Unmarshalling”
- Return values are marshalled on callee process and unmarshalled at caller process

RPC History

- SunRPC/ONC RPC (1980s, basis for NFS)
- CORBA: object-oriented middleware, hot in the 1990s
- Microsoft's DCOM and Java RMI (similar to CORBA)
- SOAP/XML-RPC: RPC using XML and HTTP (1998)
- Thrift (Facebook, 2007)
- gRPC (Google, 2015)
- REST (often with JSON)
- Ajax in web browsers

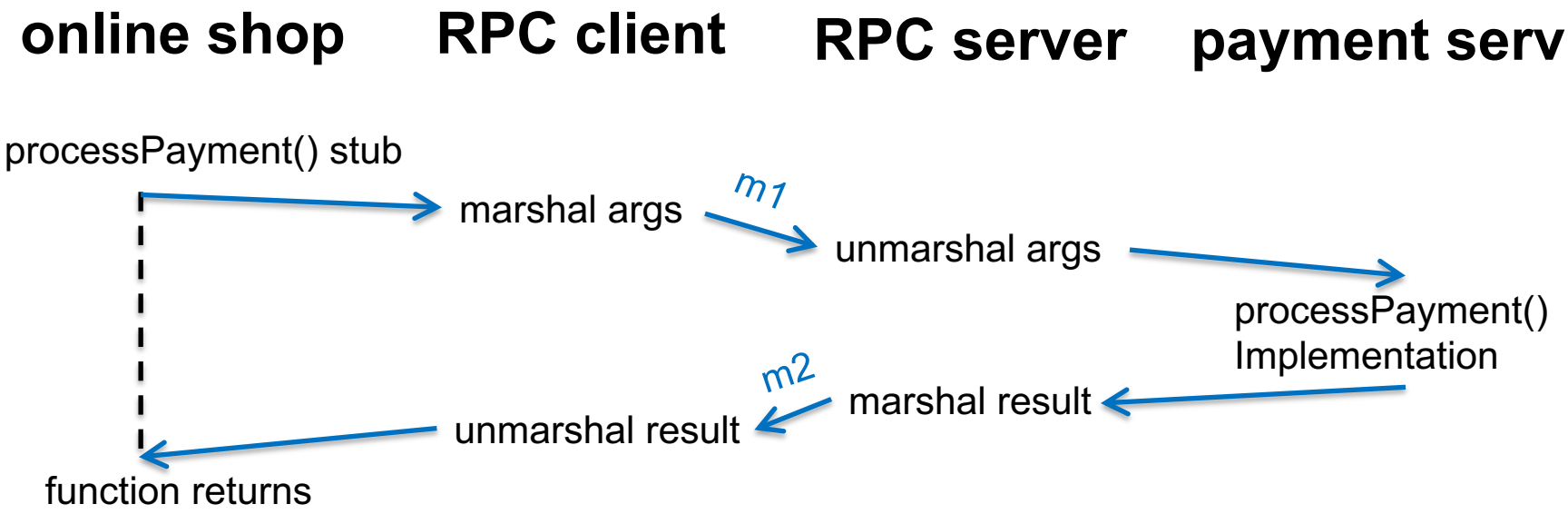
Online Payment Example



Payment Processing Code

```
// Online shop handling customer's card details
Card card = new Card();
card.setCardNumber("1234 5678 8765 4321");
card.setExpiryDate("10/2024");
card.setCVC("123");

Result result = paymentsService.processPayment(card,
3.99, Currency.GBP);

if (result.isSuccess()) {
    fulfilOrder();
}
```

RPC Implementation



m1 = { "request":"processPayment",
"card": { "number": "1234567887654321",
"expiryDate": "10/2024",
"CVC": "123" },
"amount": 3.99,
"currency": "GBP" }

m2 = { "result": "success",
"id": "XP61hHw2Rvo" }

RPC/REST in JavaScript

```
let args = {amount: 3.99, currency: 'GBP', /*...*/ };
let request = {
  method: 'POST',
  body: JSON.stringify(args), // marshaling
  headers: {'Content-Type': 'application/json'}
};
fetch('https://example.com/payments', request)
  .then((response) => {
    if (response.ok) success(response.json()); // unmarshaling
    else failure(response.status); // server error
})
  .catch((error) => {
    failure(error); // network error
});
```

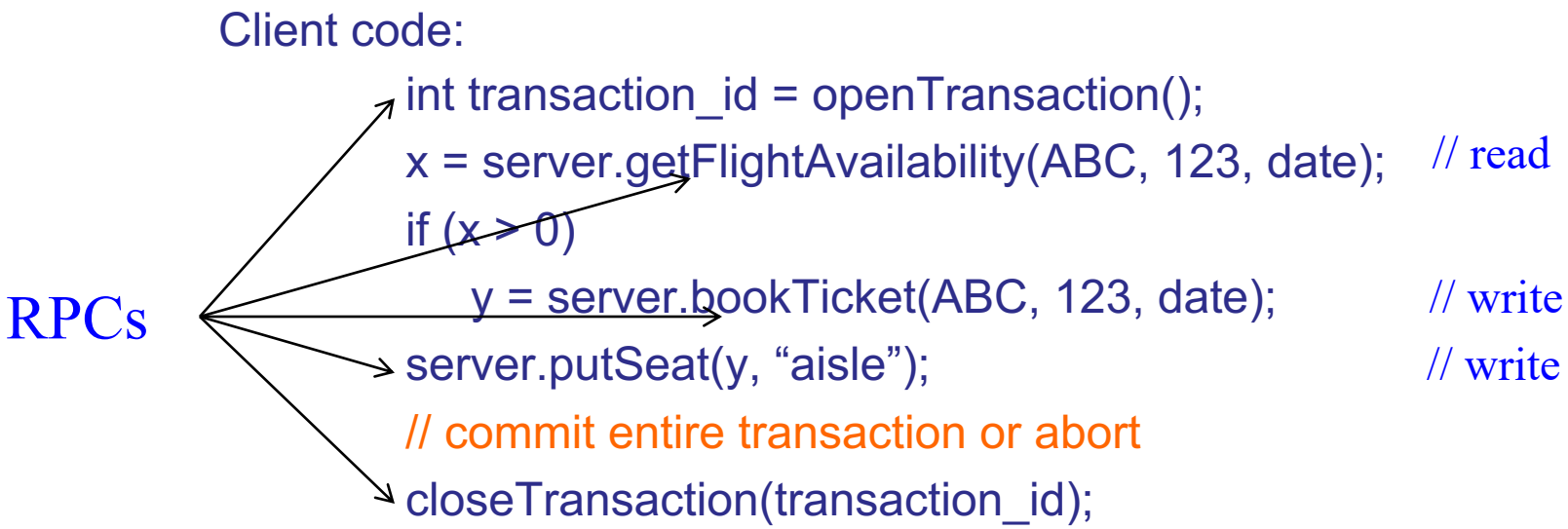
gRPC IDL

```
message PaymentRequest {
  message Card {
    required string cardNumber = 1;
    optional int32 expiryMonth = 2;
    optional int32 expiryYear = 3;
    optional int32 CVC = 4; }
  enum Currency { GBP = 1; USD = 2; }

  required Card card = 1;
  required int64 amount = 2;
  required Currency currency = 3; }

message PaymentStatus {
  required bool success = 1;
  optional string errorMessage = 2; }

service PaymentService {
  rpc ProcessPayment(PaymentRequest) returns (PaymentStatus) {} }
```

Transaction

- Series of operations executed by client
- Each operation is an RPC to a server
- Transaction either
  - completes and commits all its operations at server (commit = reflect updates on server-side objects)
  - or aborts and has no effect on server

Example: Transaction



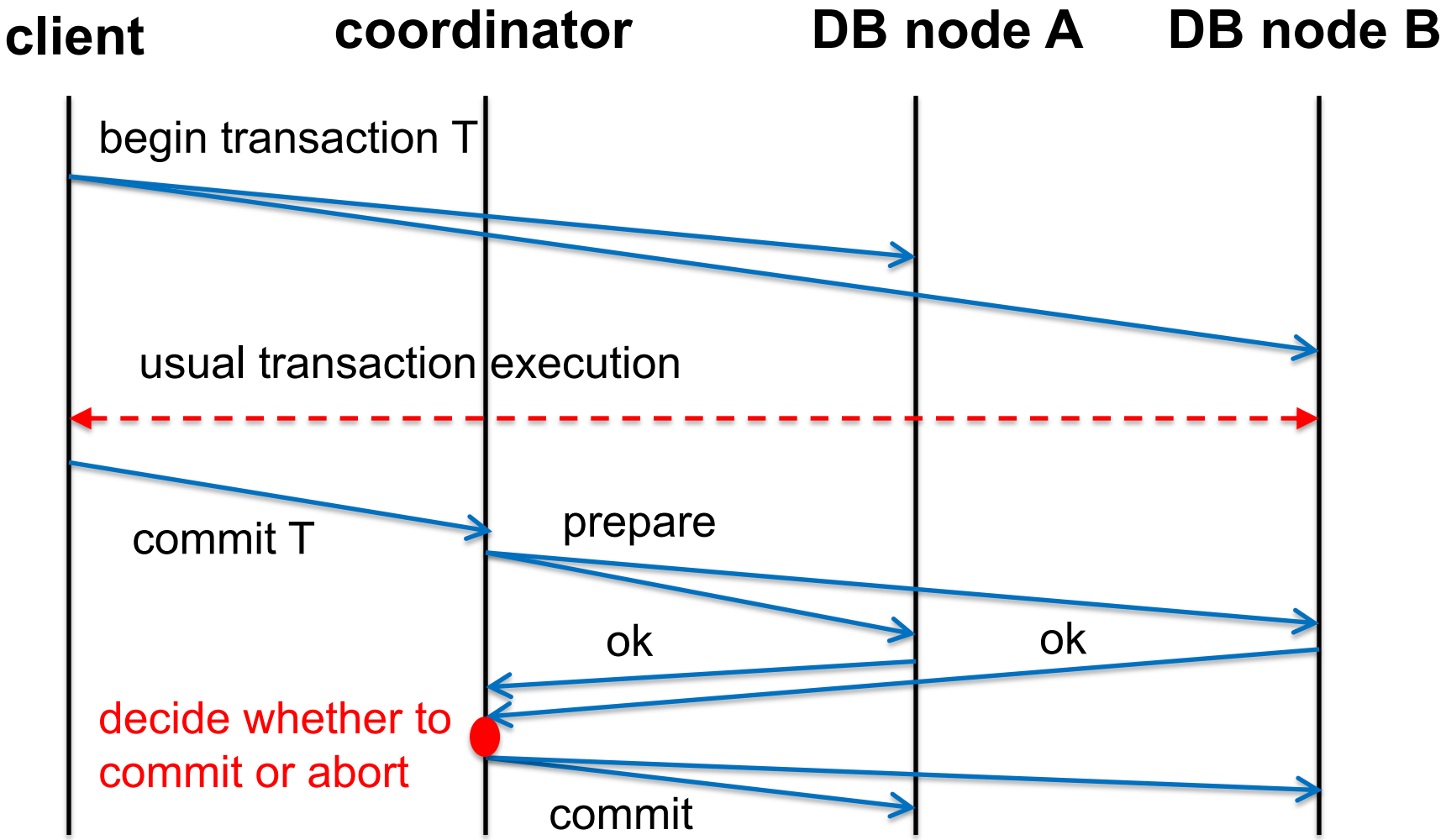
Atomicity and Isolation

- **Atomicity**: **All or nothing** principle: a transaction should either i) **complete successfully**, so its effects are recorded in the server objects; or ii) the transaction has **no effect** at all.
- **Isolation**: Need a transaction to be indivisible (**atomic**) from the point of view of other transactions
  - No access to intermediate results/states of other transactions
  - Free from interference by operations of other transactions
- But…
- Clients and/or servers might crash
- Transactions could run concurrently, i.e., with multiple clients
- Transactions may be distributed, i.e., across multiple servers

ACID Properties for Transactions

- **Atomicity**: **All or nothing**
- **Consistency**: if the server starts in a consistent state, the transaction ends the server in a consistent state.
- **Isolation**: Each transaction must be performed **without interference** from other transactions, i.e., non-final effects of a transaction must not be visible to other transactions.
- **Durability**: After a transaction has completed successfully, all its effects are saved in **permanent storage**.

Two-phase commit



Faults in Two-phase Commit

- What if the database node crashes?
- What if the coordinator crashes?
  - Coordinator writes its decision to disk
  - When it recovers, read decision from disk and send it to replicas (or abort if no decision was made before crash)
- If coordinate crashes after prepare, but before broadcasting decision, other nodes do not know how it has decided.
  - Replicas participating in transaction cannot commit or abort after responding “ok” to the prepare request
  - Algorithm is blocked until coordinator recovers

Summary

- RPC mechanism: widely used for calling functions across multiple processes
  - LPC vs. RPC
  - RPC semantics
  - RPC implementations
- Transactions
  - ACID properties
  - Two-phase commit