EPFL

The Client/Server Design Pattern

Prof. George Candea
School of Computer & Communication Sciences

Outline

same address space

- Local procedure calls (module = procedure)
- Program objects / types (module = memory object)

- Client/server architecture (different address spaces)
- *Example: RPC*

separate address spaces

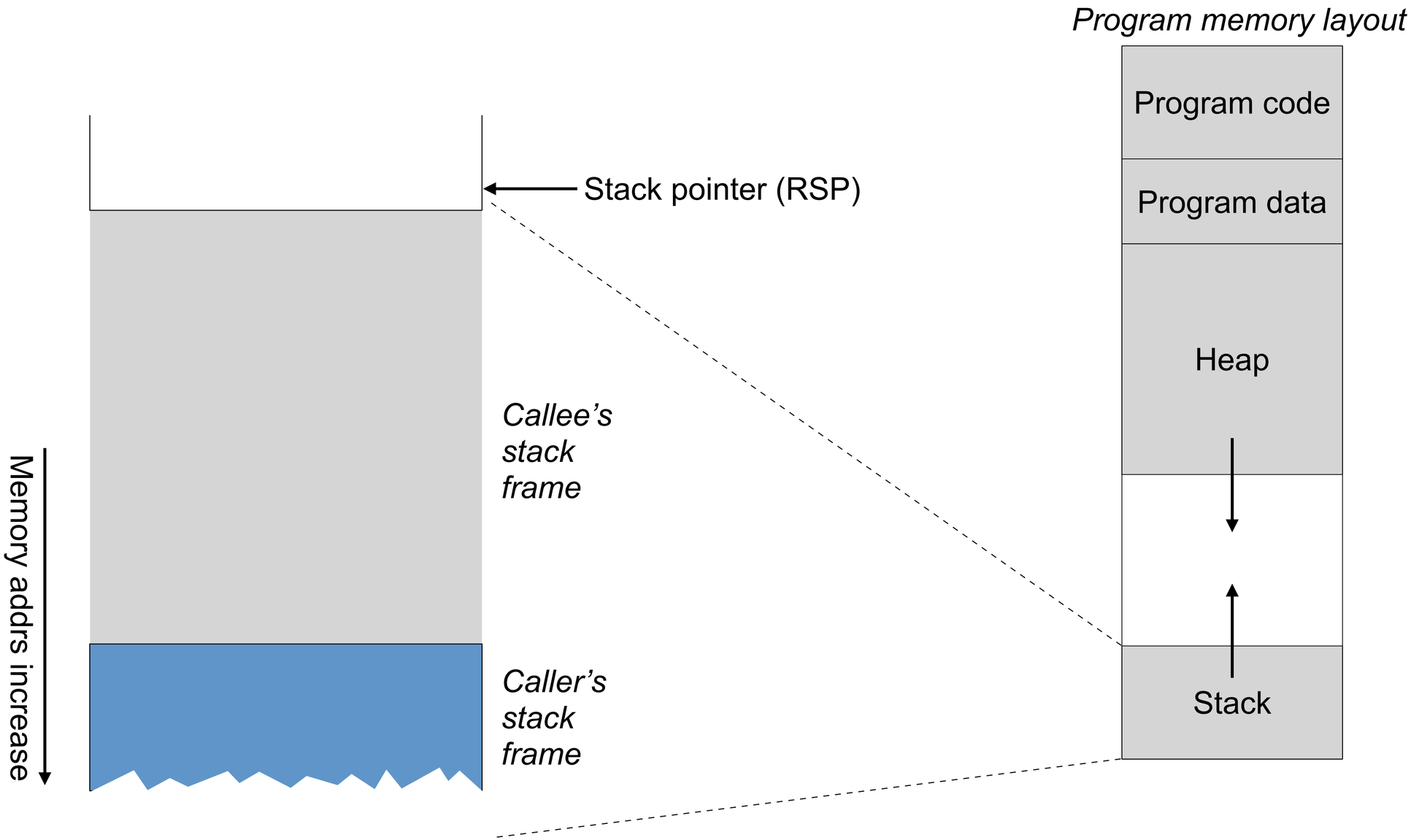
(Local) Procedure Calls

Basic mechanism for modularizing a program
(Modules = procedures)

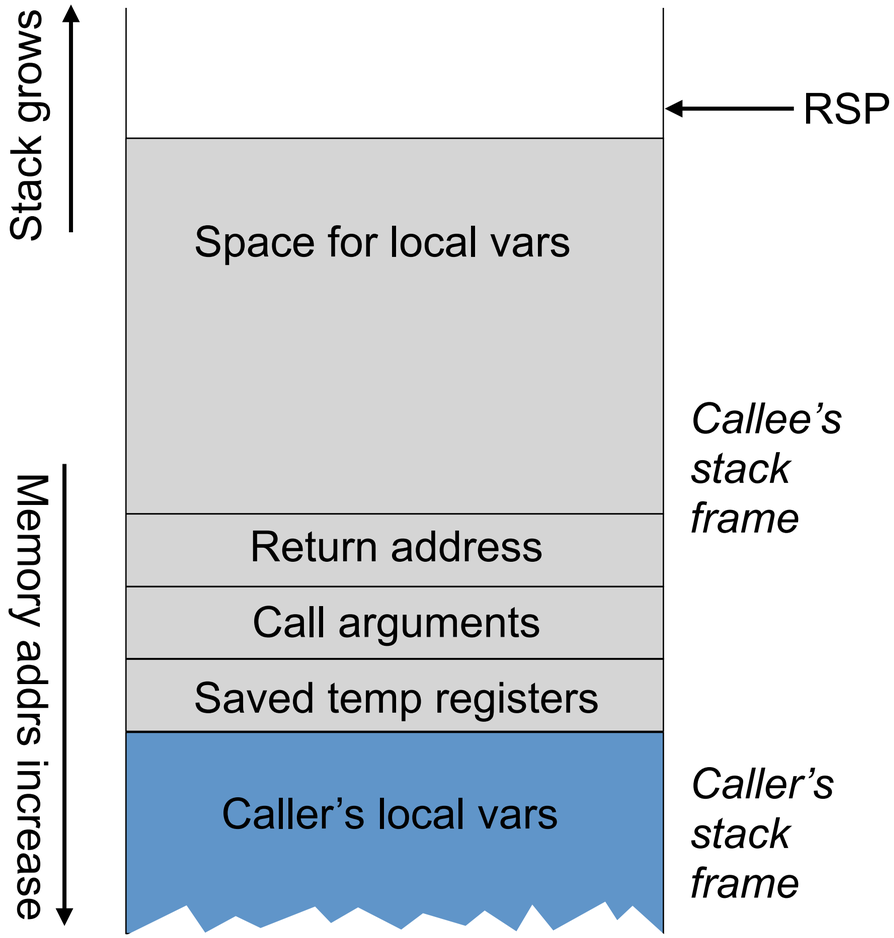
Local Procedure Calls

- Modules = procedures of the same program
- Modularization requires an inter-module communication protocol
- Protocol for procedure calls = calling convention

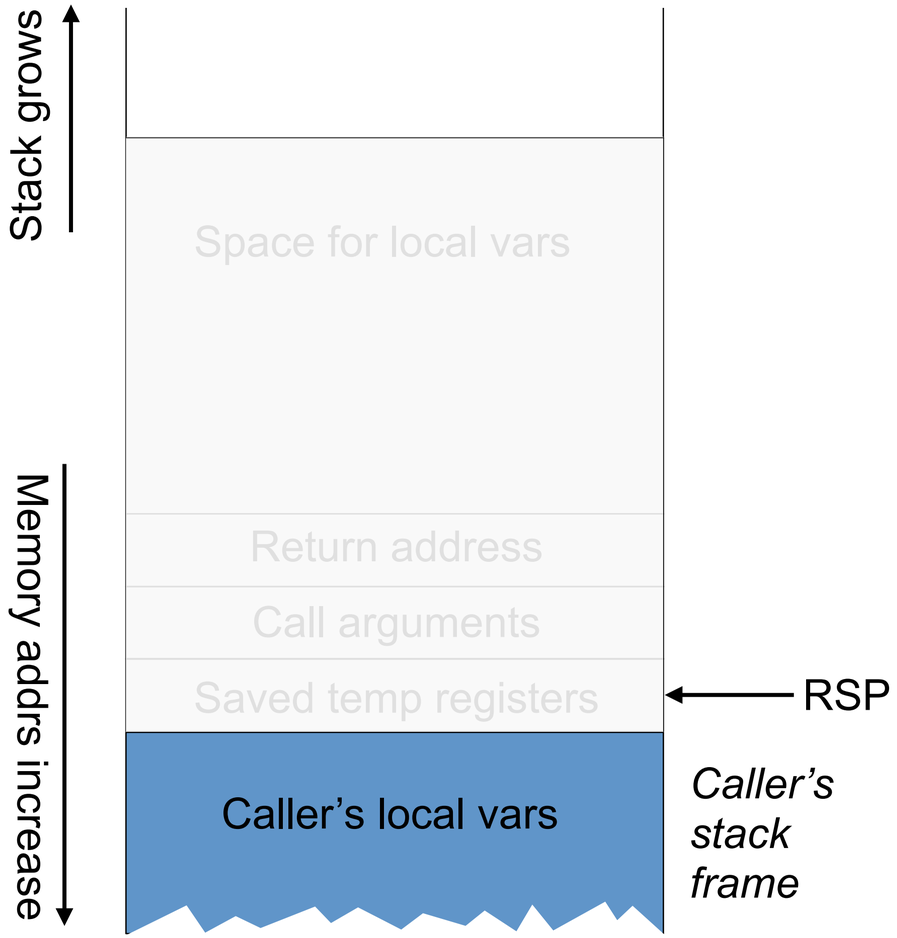
Stack-based calling convention ("interaction protocol")



Stack-based calling convention ("interaction protocol")



Stack-based calling convention ("interaction protocol")



- ABI = interface between binary modules
- Modularization
  - *Depends on programmers doing the right thing (= "soft modularization")*
  - *Compilers and runtimes help*
- Caller and callee trust each other
  - *Callee could corrupt caller's stack (e.g., buffer overflow)*
  - *Callee might return to wrong addr (e.g., stack smashing)*
  - *Callee might fail (e.g., SIGFPE due to div by zero) = "fate sharing"*
  - *Callee might leave return addr in wrong register*
  - *…*

Outline

same address space

- Local procedure calls (module = procedure)
- Program objects & types (module = memory objects)
- Client/server architecture (different address spaces)
- Example: Remote procedure calls

separate address spaces

Program Objects & Types

Strong modularization within the same address space
(Modules = objects within the program)

Program objects

```c
struct Rectangle {
    int length;
    int width;
}

int area(struct Rectangle r)
{
    return r.length * r.width;
}
```

```java
class Rectangle {

    private int length, width;

    public Rectangle(int l, int b)
    {
        length = l;
        width = b;
    }

    public int area()
    {
        return length * width;
    }
}
```

Program objects

Data

Behavior

```
struct Rectangle {
    int length;
    int width;
}

int area(struct Rectangle r)
{
    return r.length * r.width;
}
```

vs.

Encapsulation

```
class Rectangle {

    private int length, width;

    public Rectangle(int l, int b)
    {
        length = l;
        width = b;
    }

    public int area()
    {
        return length * width;
    }
}
```

Data + Behavior inseparable

Objects & type safety = stronger intra-program modularity

- Untyped languages
- Weakly typed languages (e.g., C)
  - *Have types, but can change (e.g., explicitly cast data from one type to another)*
- Strongly typed languages (e.g., Lisp)
  - *Each chunk of memory has well defined type, no Object or void*
  - *Python, C#, C++, Rust, ... might qualify*
- Ensuring type safety
  - *Static (Rust, Haskell) vs. dynamic (Python, Ruby)*

Soft vs. enforced modularization

- Programmers are humans
  - *Trusting gives you at best a "soft" modularization*
- Better to trust compilers, runtimes, libraries, operating systems, …
  - *E.g., modularize using Docker-style containers (OS-level virtualization)*
  - *Lower layers are widely used and robust (even though they too are buggy…)*
- Better to trust hardware
  - *Cheap way to (sort of) do this: modularize using virtual machines*
  - *Widely used and robust (even though it too is buggy…)*

The lower the layer where modularity is enforced, the stronger the modularity

Outline

same address space

- Local procedure calls (module = procedure)
- Program objects & types (module = memory object)

- Client/server architecture (different address spaces)
- Example: Remote procedure calls

separate address spaces

Clients/Servers Interacting via Messages

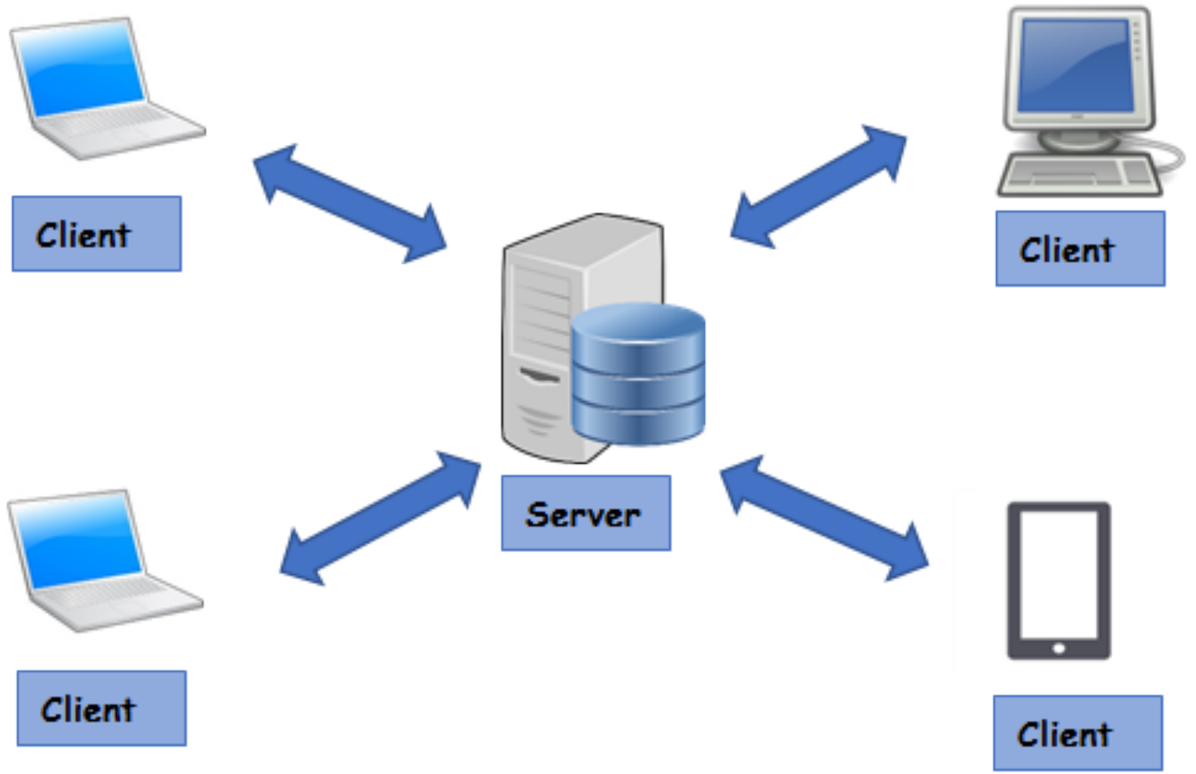
Modularization across different address spaces

Splitting into Clients and Servers

- Is the foundation for many system architecture patterns
  - *event-driven, microservices (formerly SOA), action–domain–responder (e.g., MVVM), multi-tiered, peer-to-peer, publish-subscribe, etc.*
- Key ideas
  - *place modules in separate, strongly isolated domains, and have them communicate via messages*
  - *messages typically need to be marshalled/unmarshalled for send/receive*

Physical (and virtual) servers

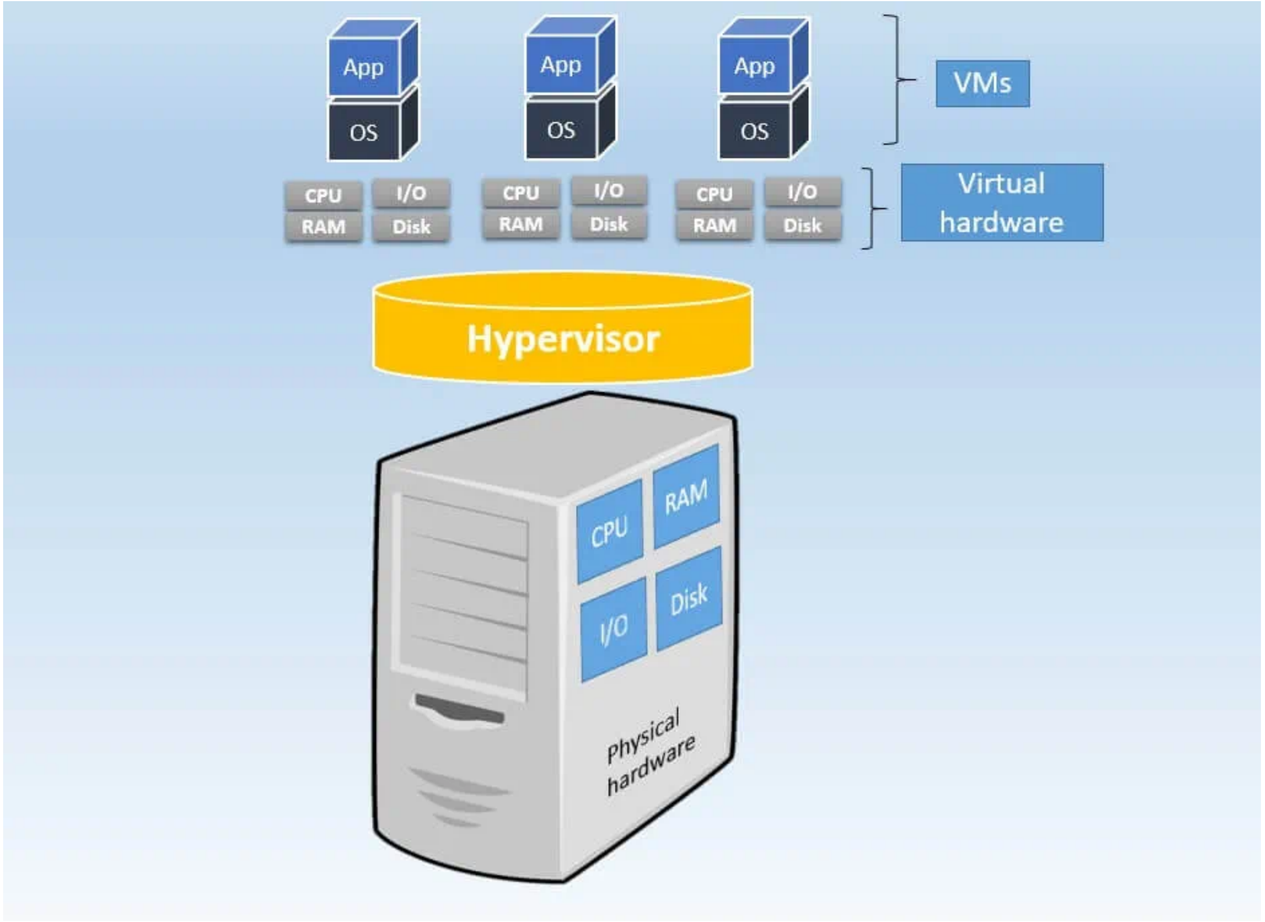
- Rely on physics
- Reduce fate sharing
- Improve encapsulation



https://www.omnisci.com/technical-glossary/client-server

Physical (and virtual) servers

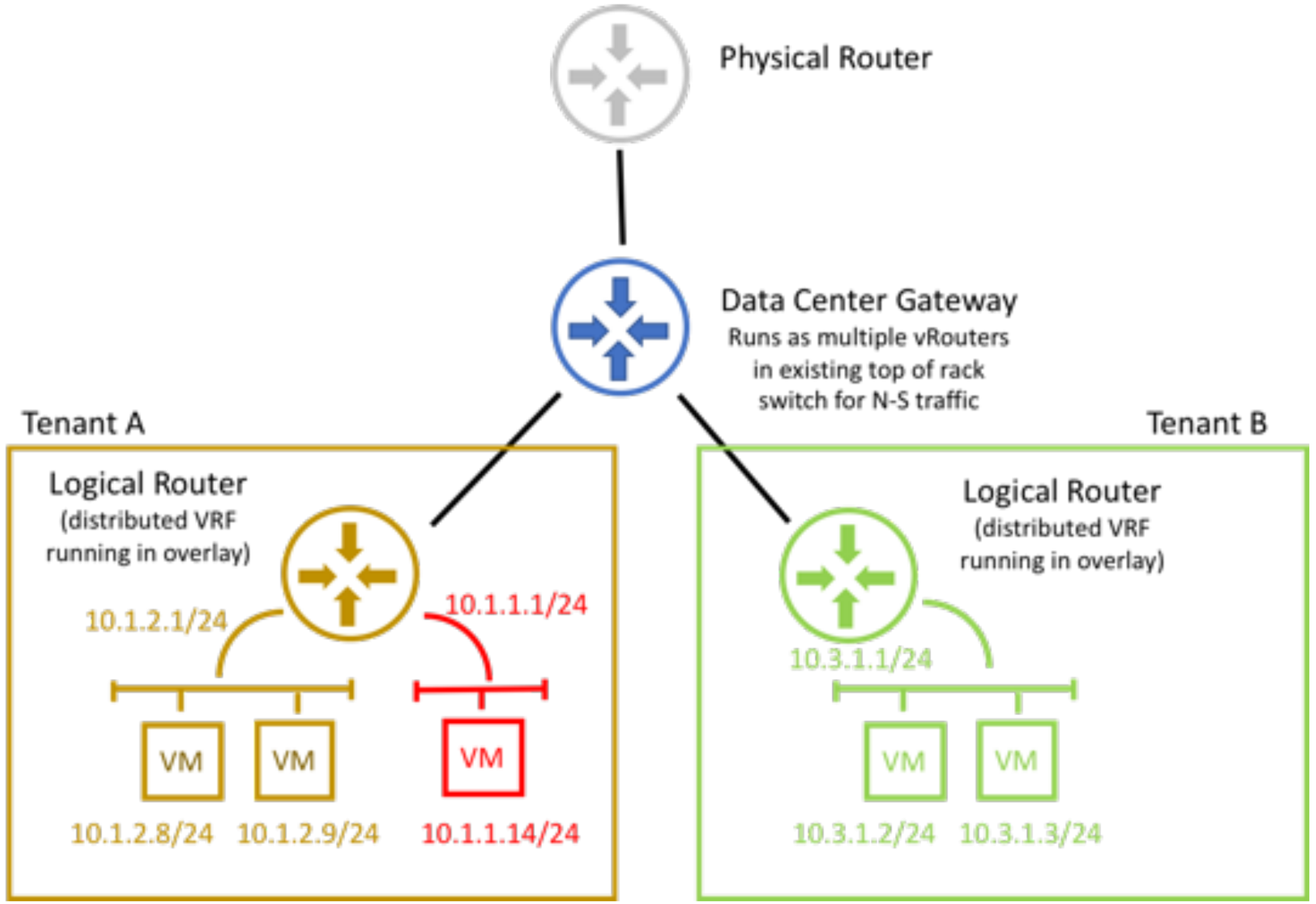
- Rely on physics
- Reduce fate sharing
- Improve encapsulation



https://www.nakivo.com/blog/wp-content/uploads/2018/12/Virtual-server-architecture.webp

Physical (and virtual) servers

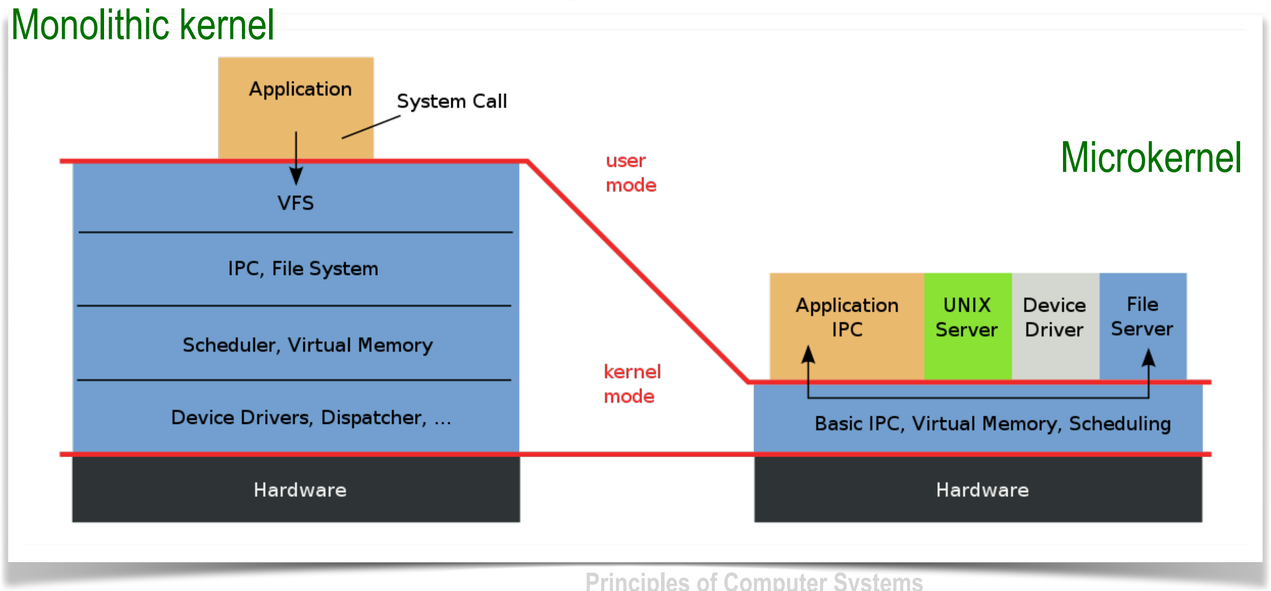
- Rely on physics
- Reduce fate sharing
- Improve encapsulation



https://www.pluribusnetworks.com/blog/what-is-network-segmentation/

Microkernels

- An exercise in modularization of otherwise monolithic kernels
  - *Liedtke's minimality principle*
- Servers = trusted intermediaries
  - *Essentially daemon programs with some extra privileges*
  - *e.g., can access physical memory that would otherwise be off-limits*



Microkernels

- An exercise in modularization of otherwise monolithic kernels
  - *Liedtke's minimality principle*
- Servers = trusted intermediaries
  - *Essentially daemon programs with some extra privileges*
  - *e.g., can access physical memory that would otherwise be off-limits*
- Talks to servers over IPC (inter-process communication)
  - *Instead of syscalls in monolithic kernels*
- How is fate sharing? How is encapsulation?

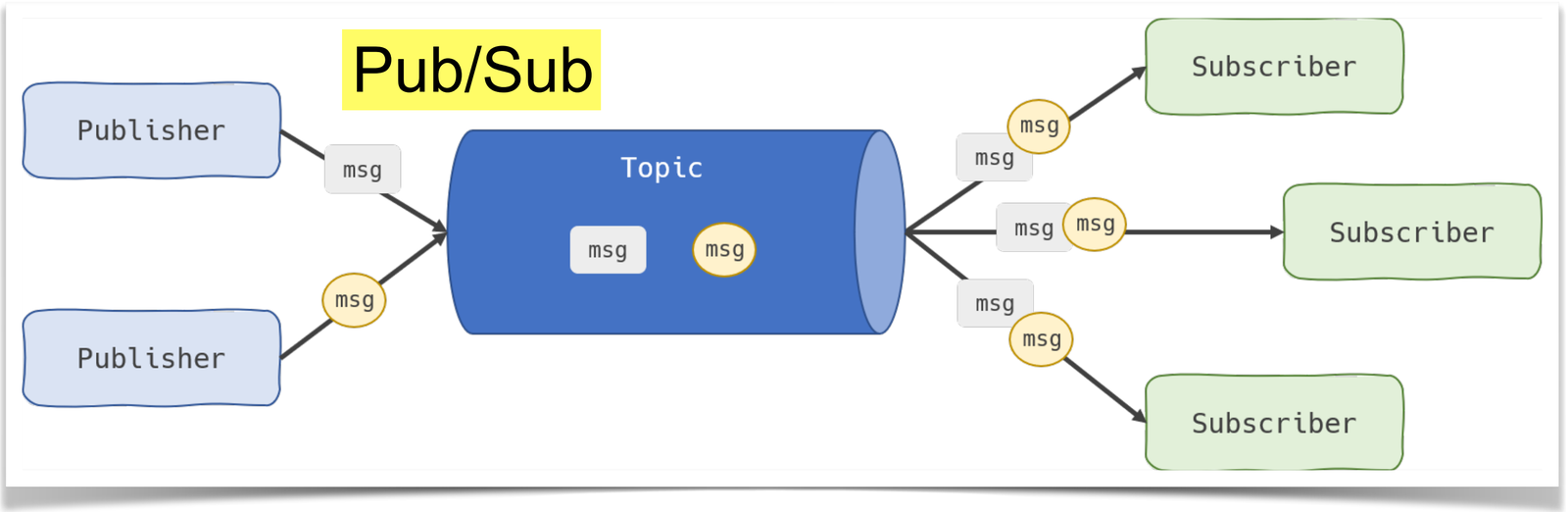
Benefits of Client/Server

- Narrow channels for error propagation
  - *Isolation between "caller" and "callee"*
- Decoupling
  - *Can fail independently —> the opposite of "fate sharing"*
  - *Rely on timeouts to infer remote failure*
- Forcing function to document interfaces

Drawbacks of Client/Server

- Marshalling/unmarshalling messages incurs overheads
- Unnatural interaction between modules
- Semantic coupling may render functional decoupling moot
  - *E.g., caller cannot make progress without an answer*

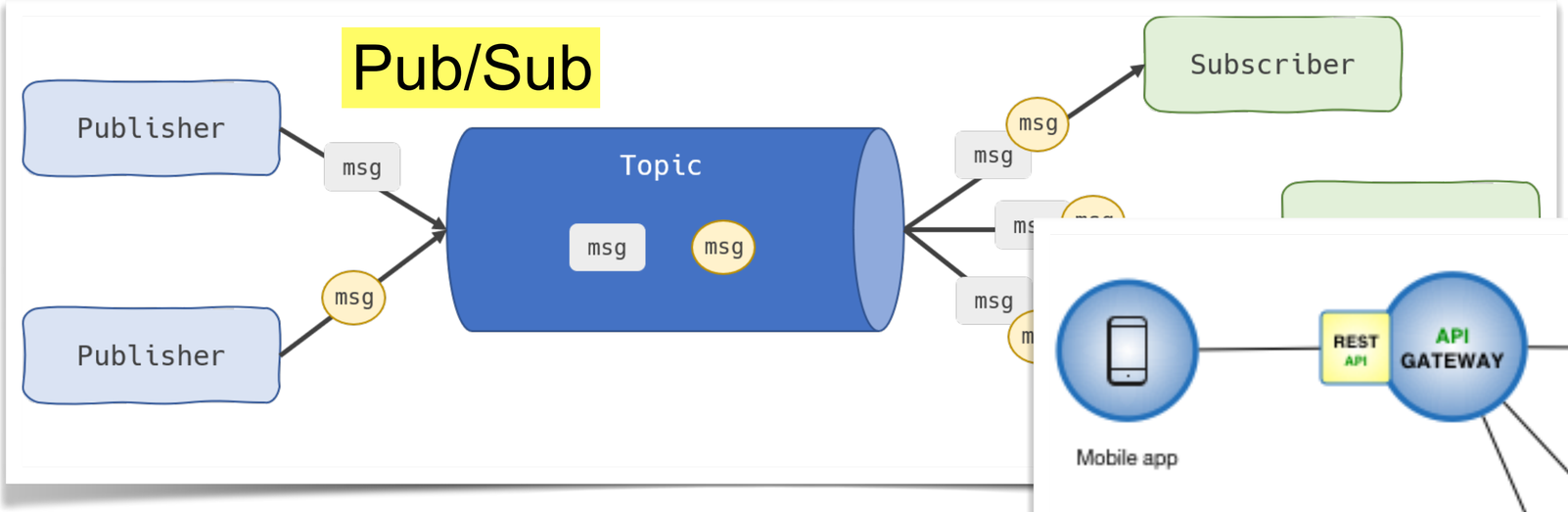
A couple of examples of client/server architectures

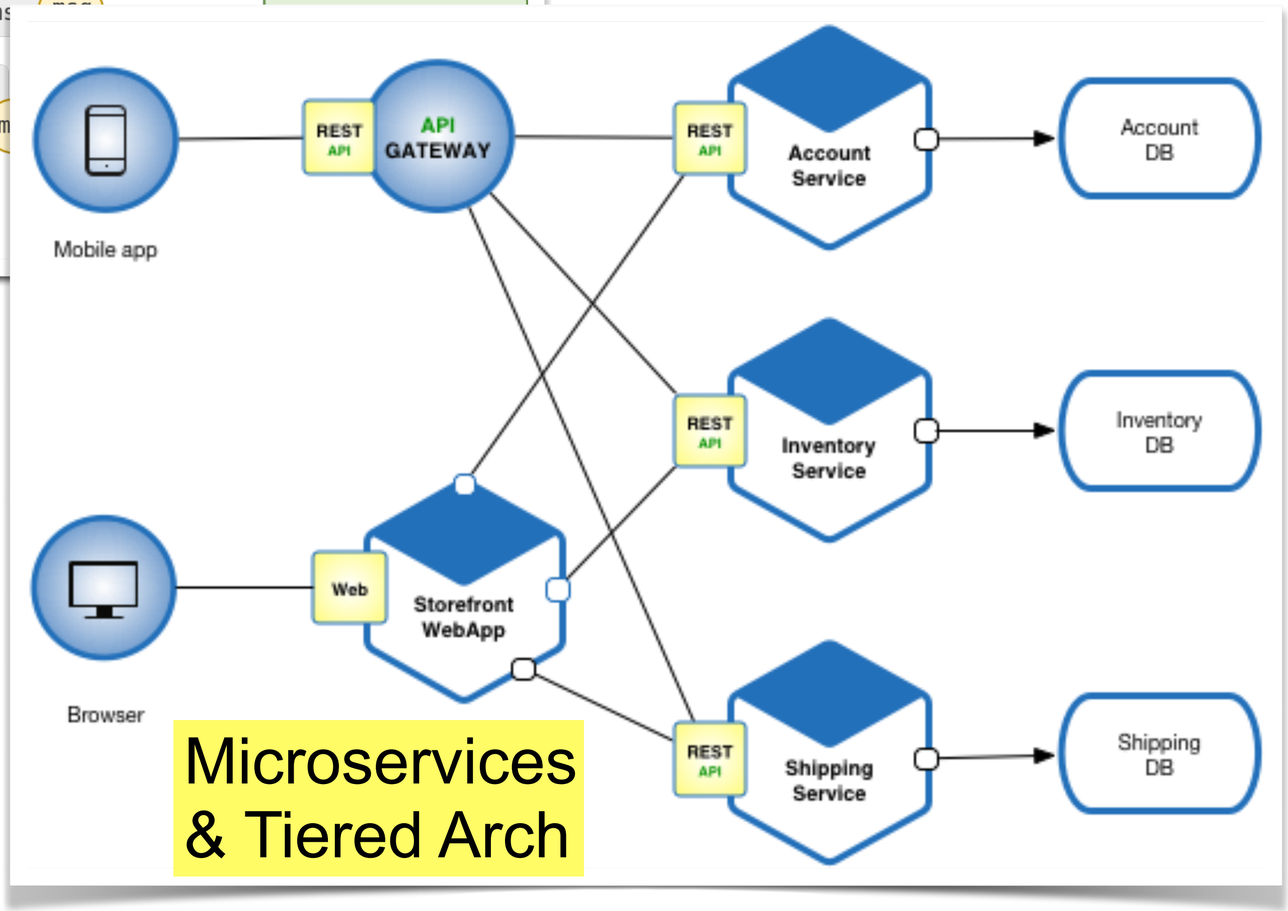


https://dashbird.io/knowledge-base/well-architected/pub-sub-messaging/

https://microservices.io/i/Microservice_Architecture.png

A couple of examples of client/server architectures





https://dashbird.io/knowledge-base/well-architected/pub-sub-messaging/

https://microservices.io/i/Microservice_Architecture.png

Outline

same address space

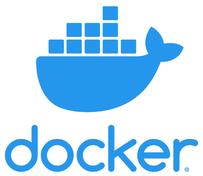
- Local procedure calls (module = procedure)
- Program objects & types (module = memory object)
- Client/server architecture (different address spaces)
- Example: Remote procedure calls

separate address spaces

Remote Procedure Calls (RPC)

Get benefits of client/server organization
with the comfort of a procedure call

RPC is everywhere



Mechanics of RPC

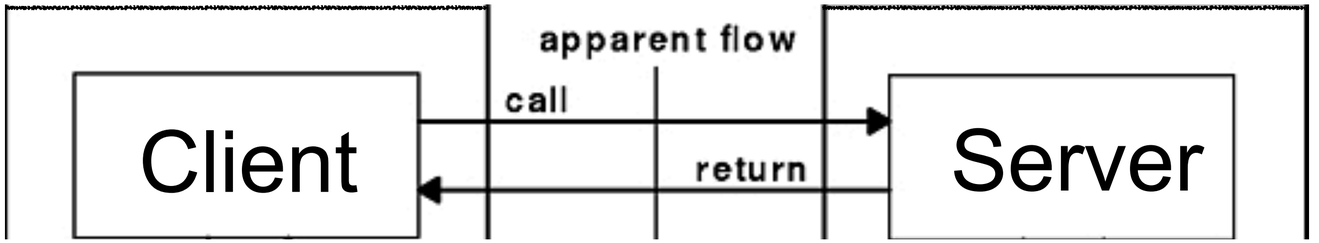


Image courtesy of https://www.ibm.com/support/knowledgecenter

Mechanics of RPC

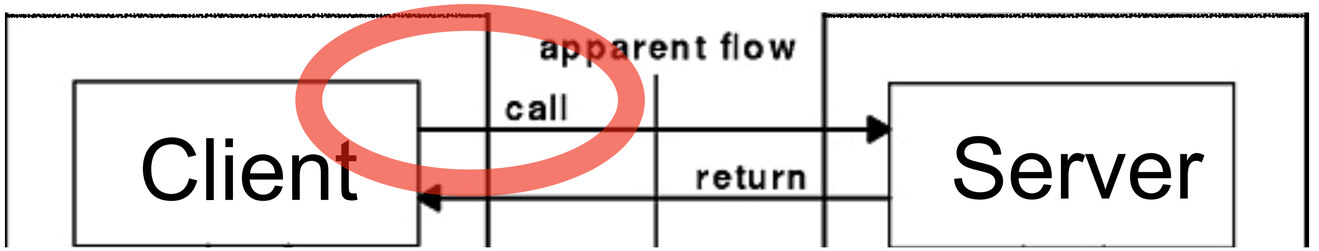


Image courtesy of https://www.ibm.com/support/knowledgecenter

Mechanics of RPC

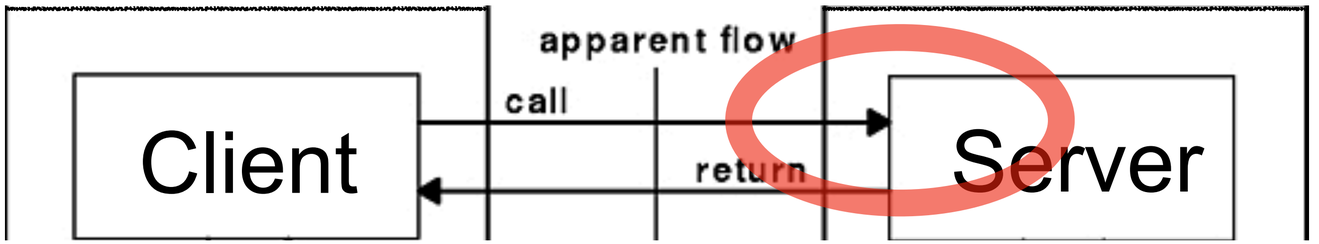


Image courtesy of https://www.ibm.com/support/knowledgecenter

Mechanics of RPC

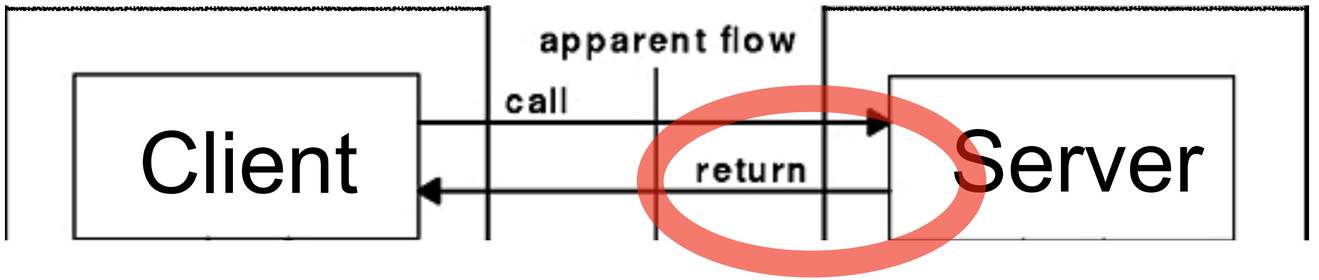


Image courtesy of https://www.ibm.com/support/knowledgecenter

Mechanics of RPC

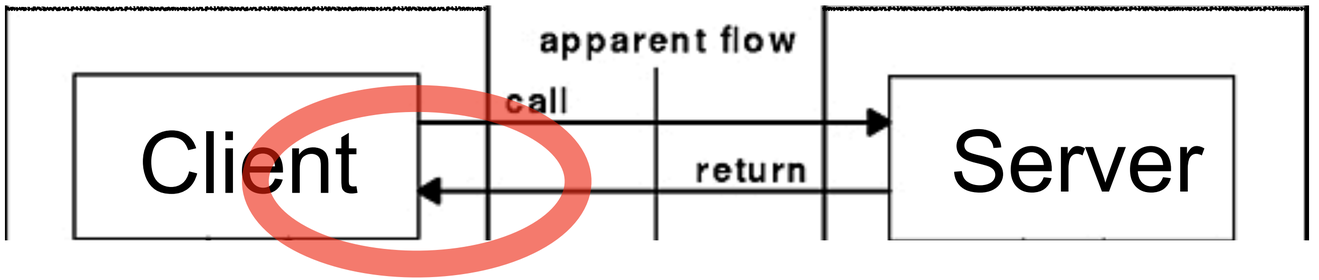


Image courtesy of https://www.ibm.com/support/knowledgecenter

Mechanics of RPC

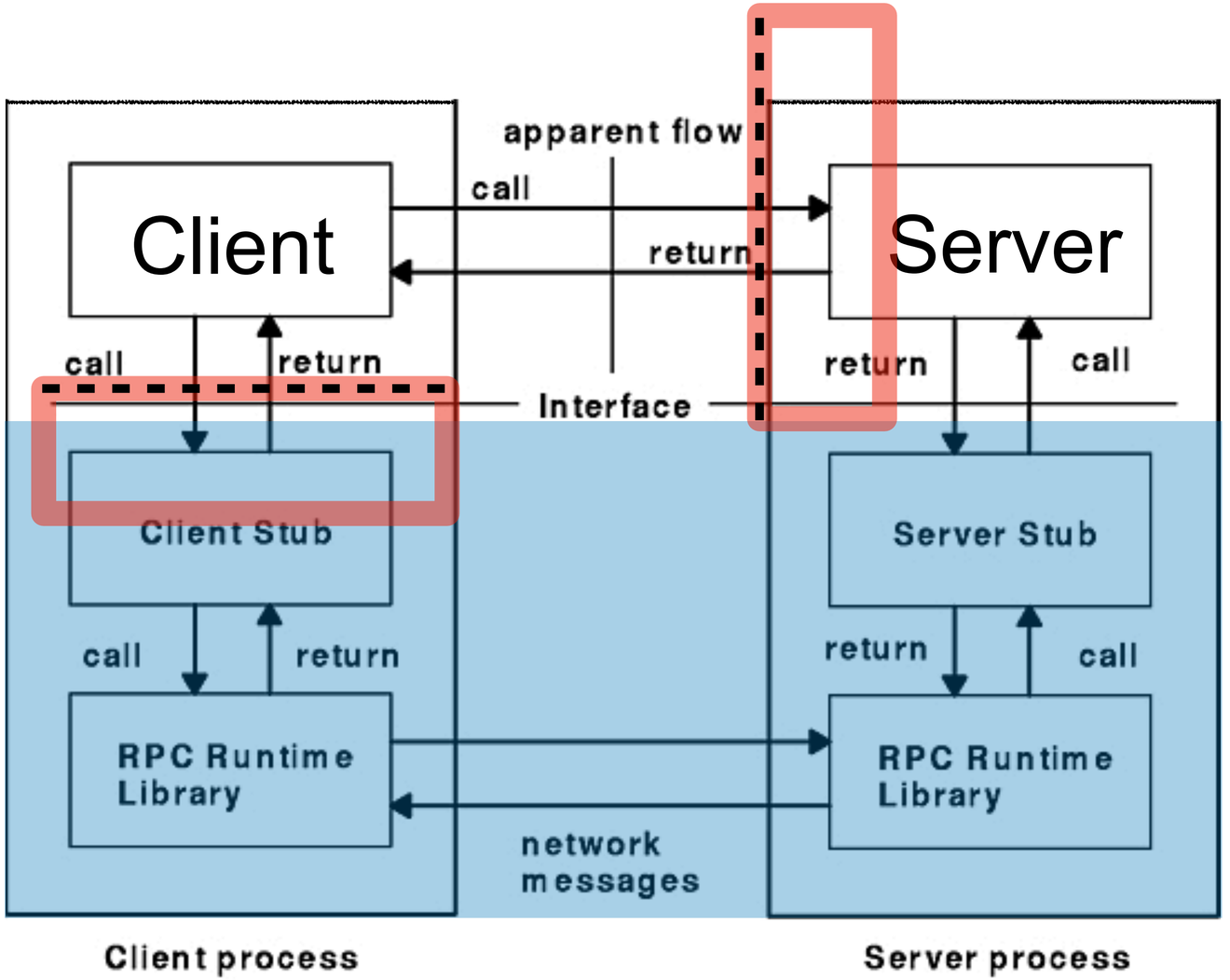
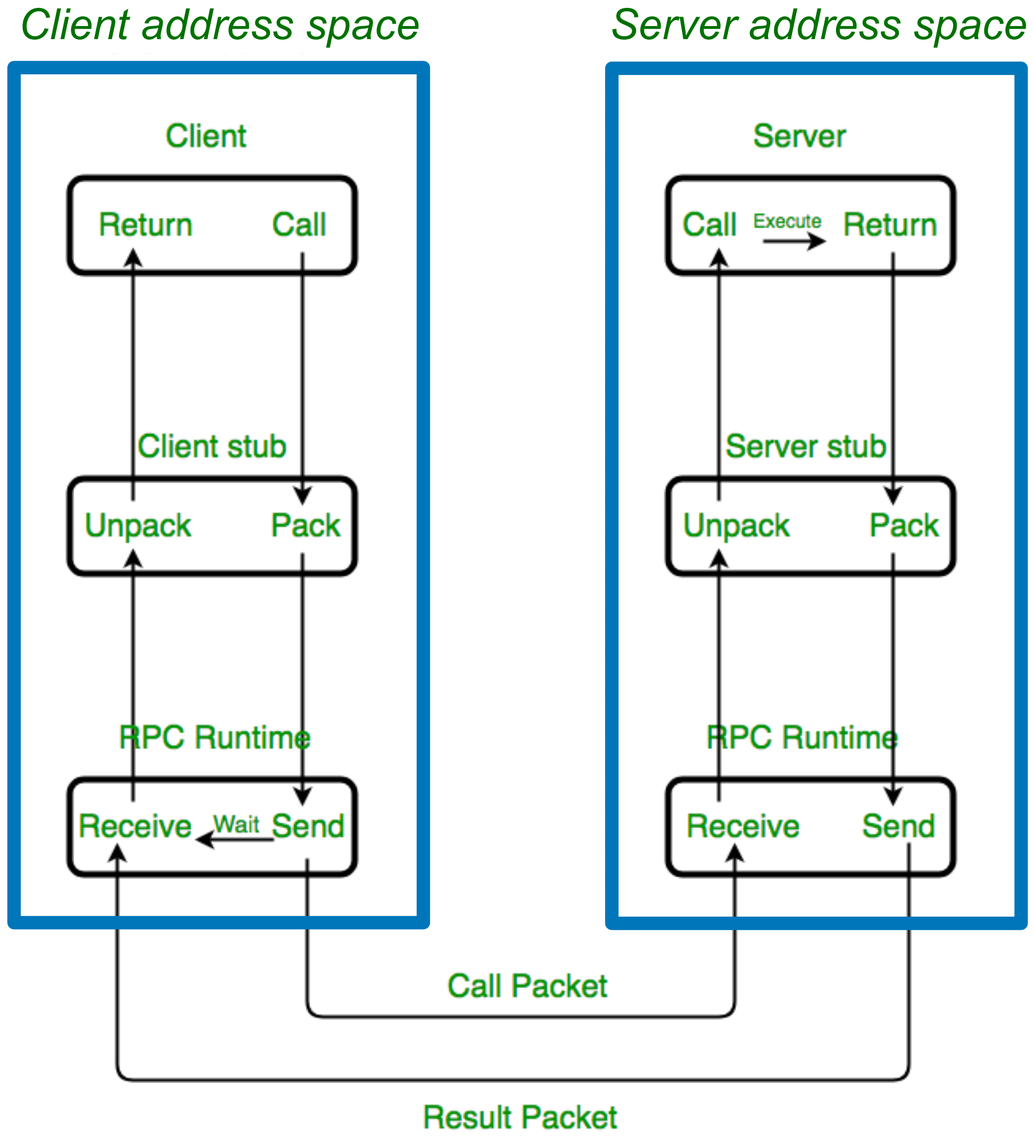


Image courtesy of https://www.ibm.com/support/knowledgecenter

Mechanics of RPC



Local procedure call

Parameters -> message

Message -> packet

Local procedure invocation

Message -> parameters

Packet -> message

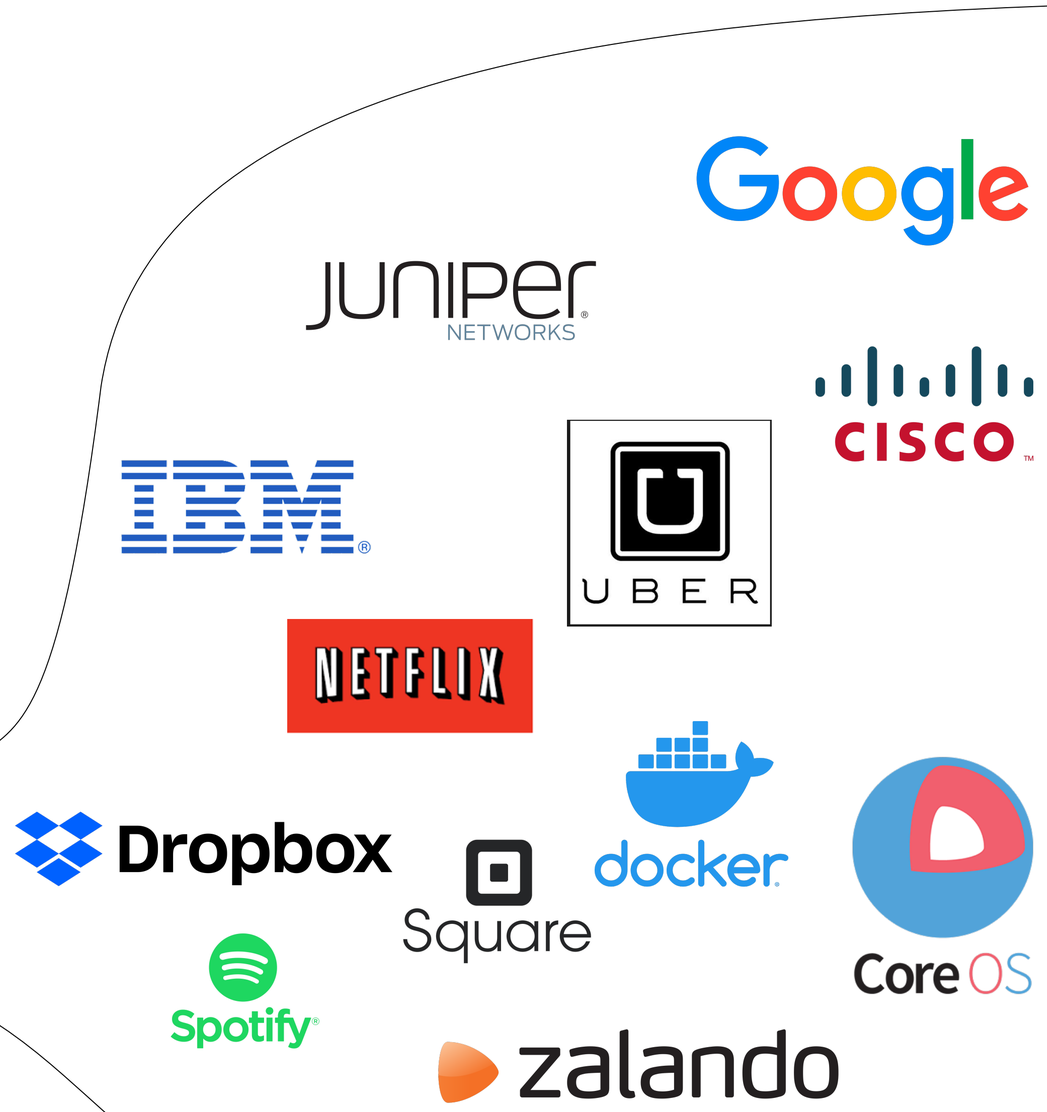
Image courtesy of https://www.geeksforgeeks.org/remote-procedure-call-rpc-in-operating-system/

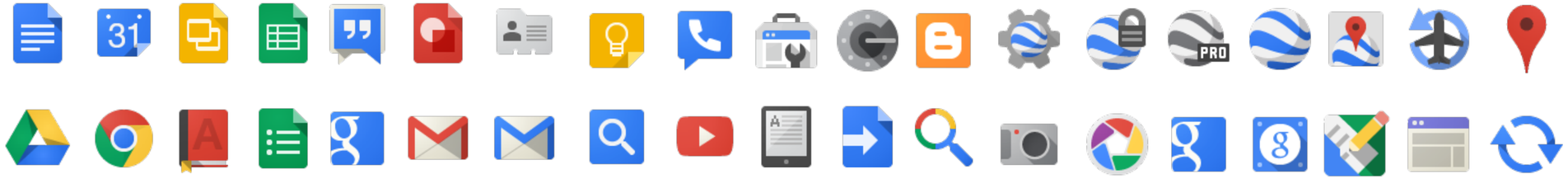
Examples of RPC systems

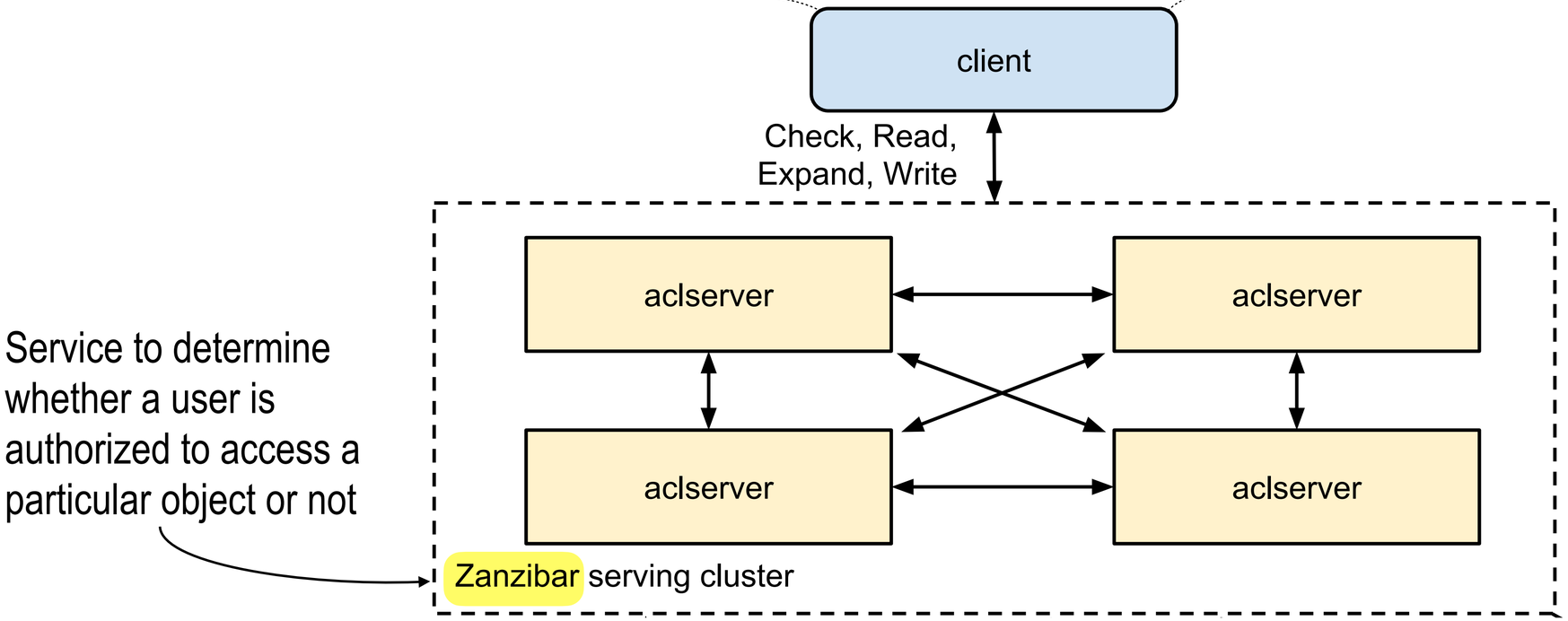
- NFS *and*  Google Cloud Filestore
- Java RMI *and* Google Web Toolkit
- Go's `rpc` package
- Cassandra, HBase, Couchbase
- Apache Thrift
- gRPC (uses Google Protocol Buffers IDL)
  - *microservices, mobile, real-time, IoT, ...*

Examples of RPC systems

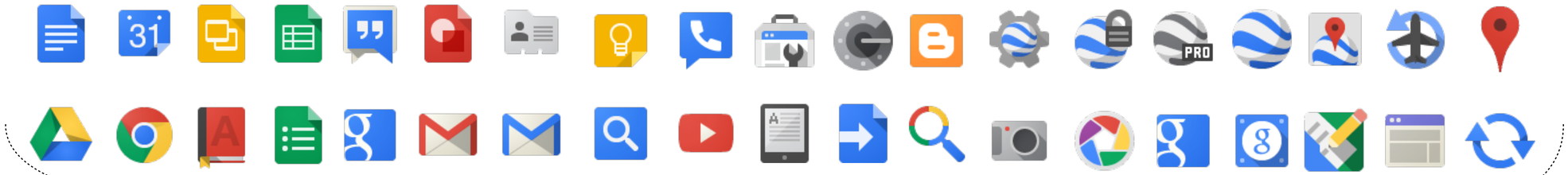
- NFS *and* Google Cloud Filestore
- Java RMI *and* Google Web Toolkit
- Go's `rpc` package
- Cassandra, HBase, Couchbase
- Apache Thrift
- gRPC (uses Google Protocol Buffers IDL)
  - *microservices, mobile, real-time, IoT, ...*

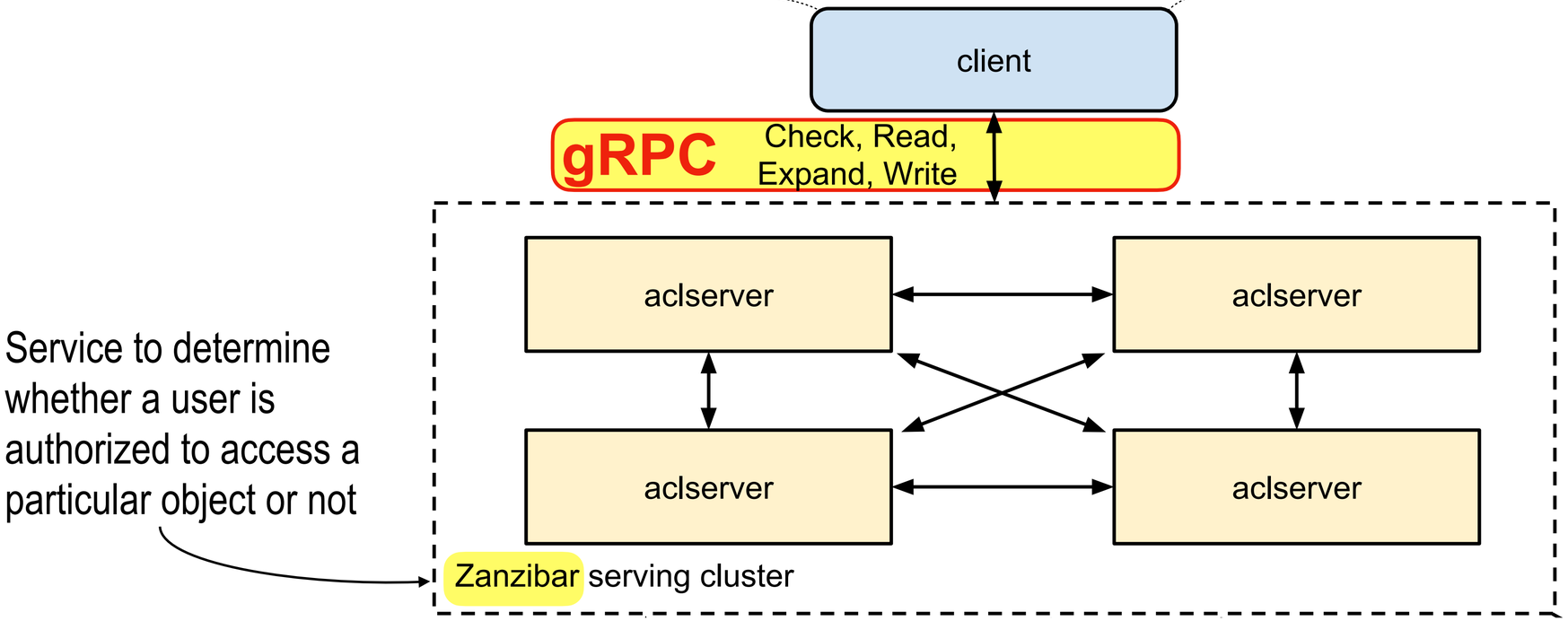



31
PRO



Service to determine whether a user is authorized to access a particular object or not





Service to determine whether a user is authorized to access a particular object or not

client

gRPC Check, Read, Expand, Write

Service to determine whether a user is authorized to access a particular object or not

aclserver

aclserver

aclserver

aclserver

Zanzibar serving cluster

read/write

read/write

append

Externally consistent transactional database

namespace configs

namespace 1 relation tuples

...

namespace N relation tuples

changelog

Spanner global database system

client

gRPC Check, Read, Expand, Write

aclserver

aclserver

aclserver

aclserver

Zanzibar serving cluster

Service to determine whether a user is authorized to access a particular object or not

gRPC read/write read/write append

namespace configs

namespace 1 relation tuples

...

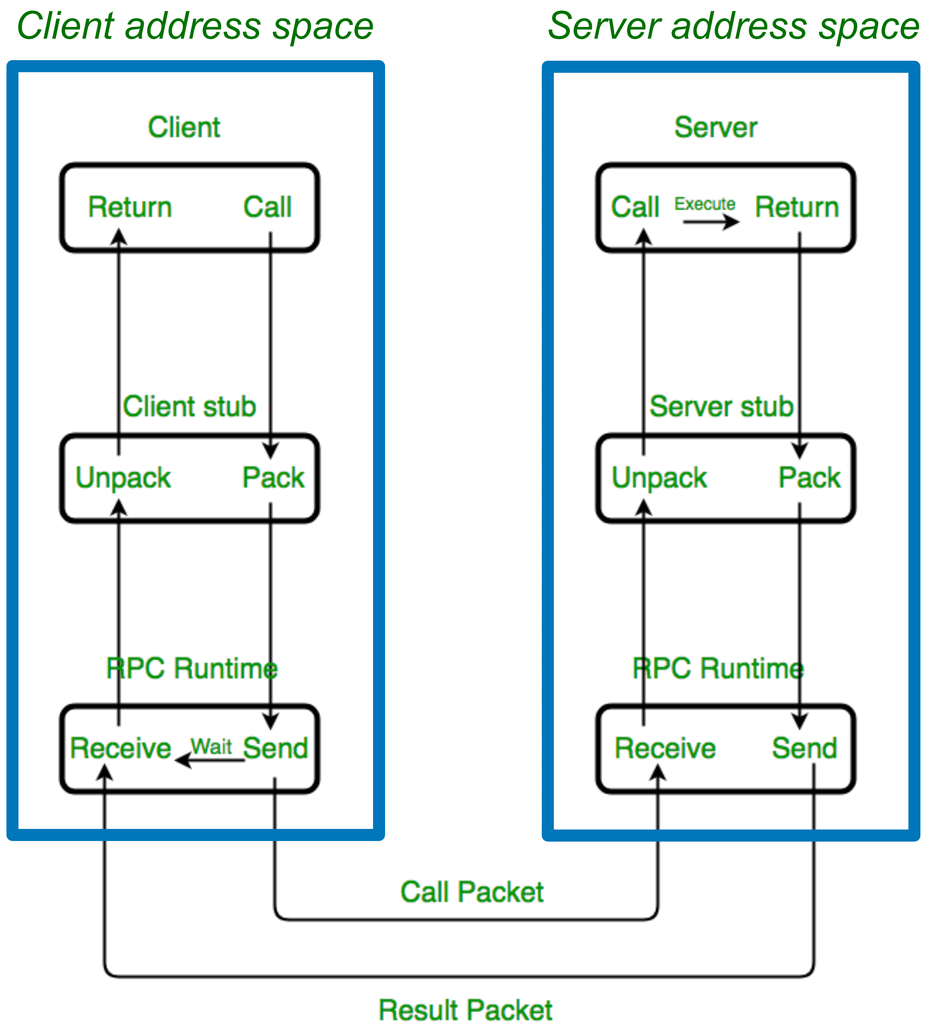
namespace N relation tuples

changelog

Spanner global database system

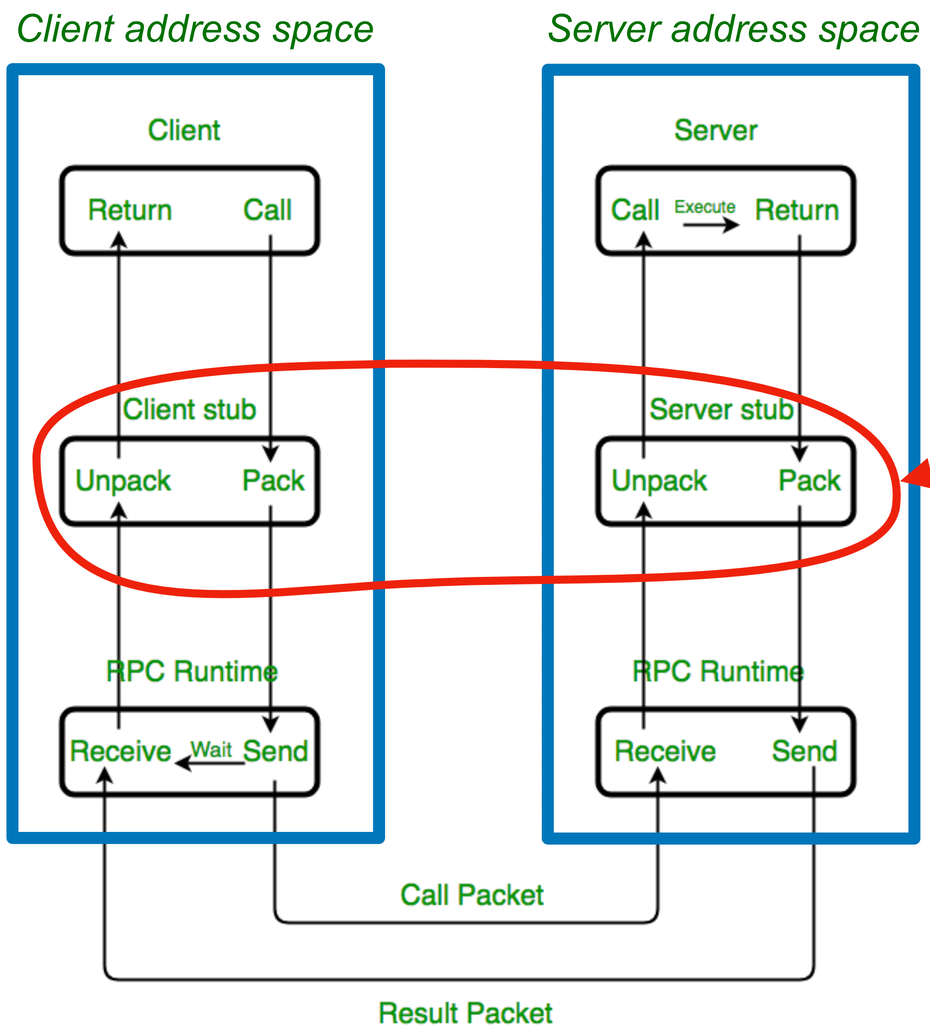
Externally consistent transactional database

Workflow for writing RPC-based systems



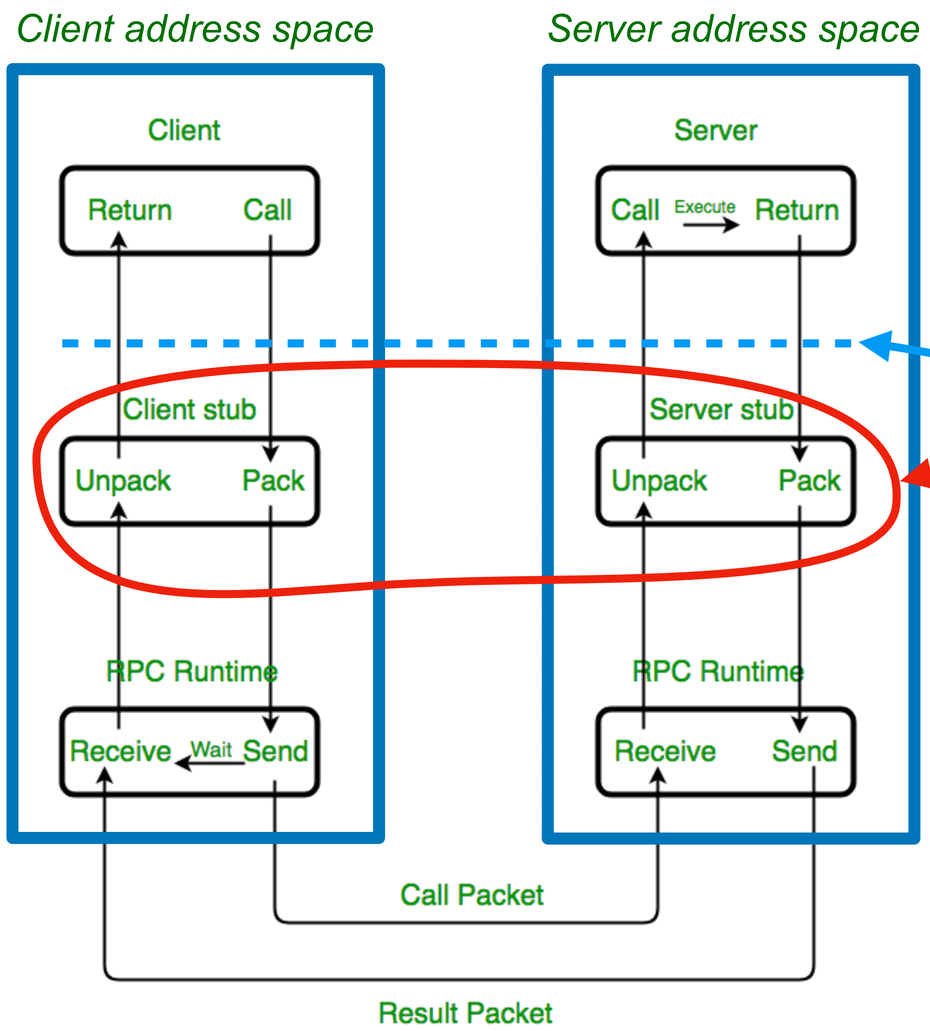
- Define the service in an IDL file

Workflow for writing RPC-based systems



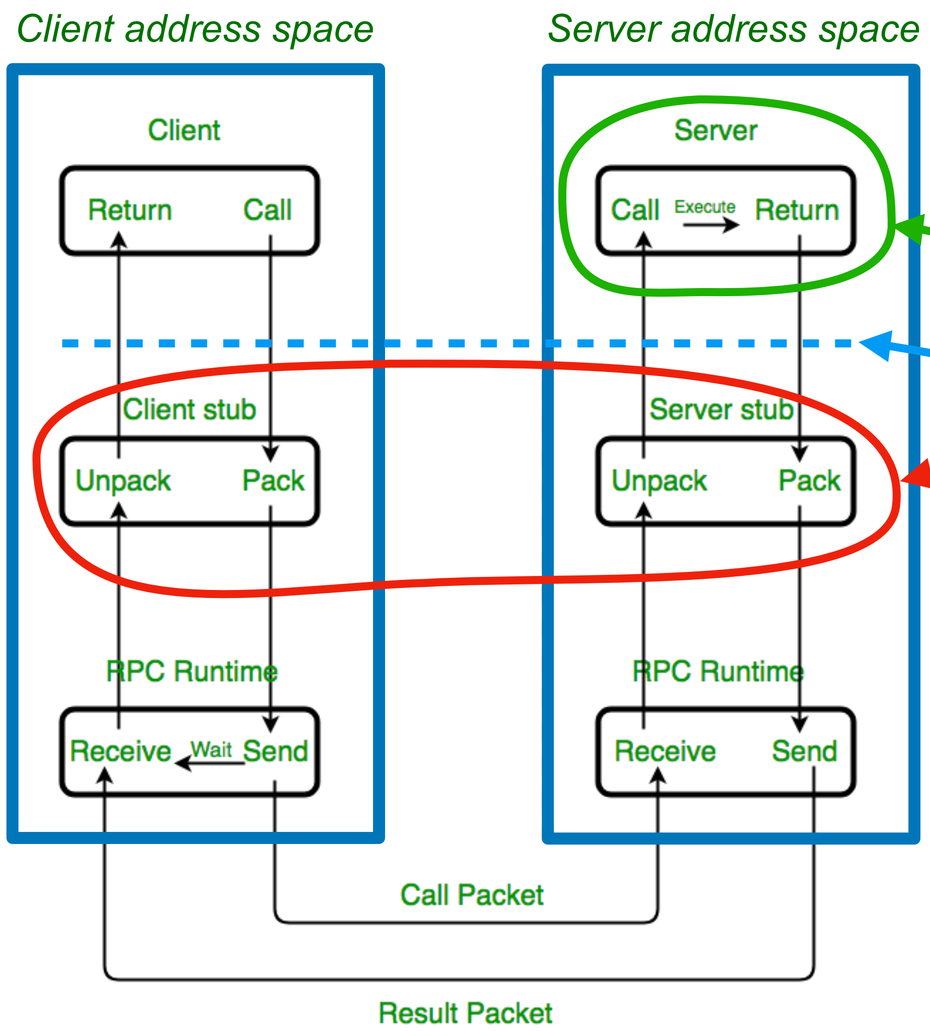
- Define the service in an IDL file
- Generate message implementations using the IDL compiler

Workflow for writing RPC-based systems



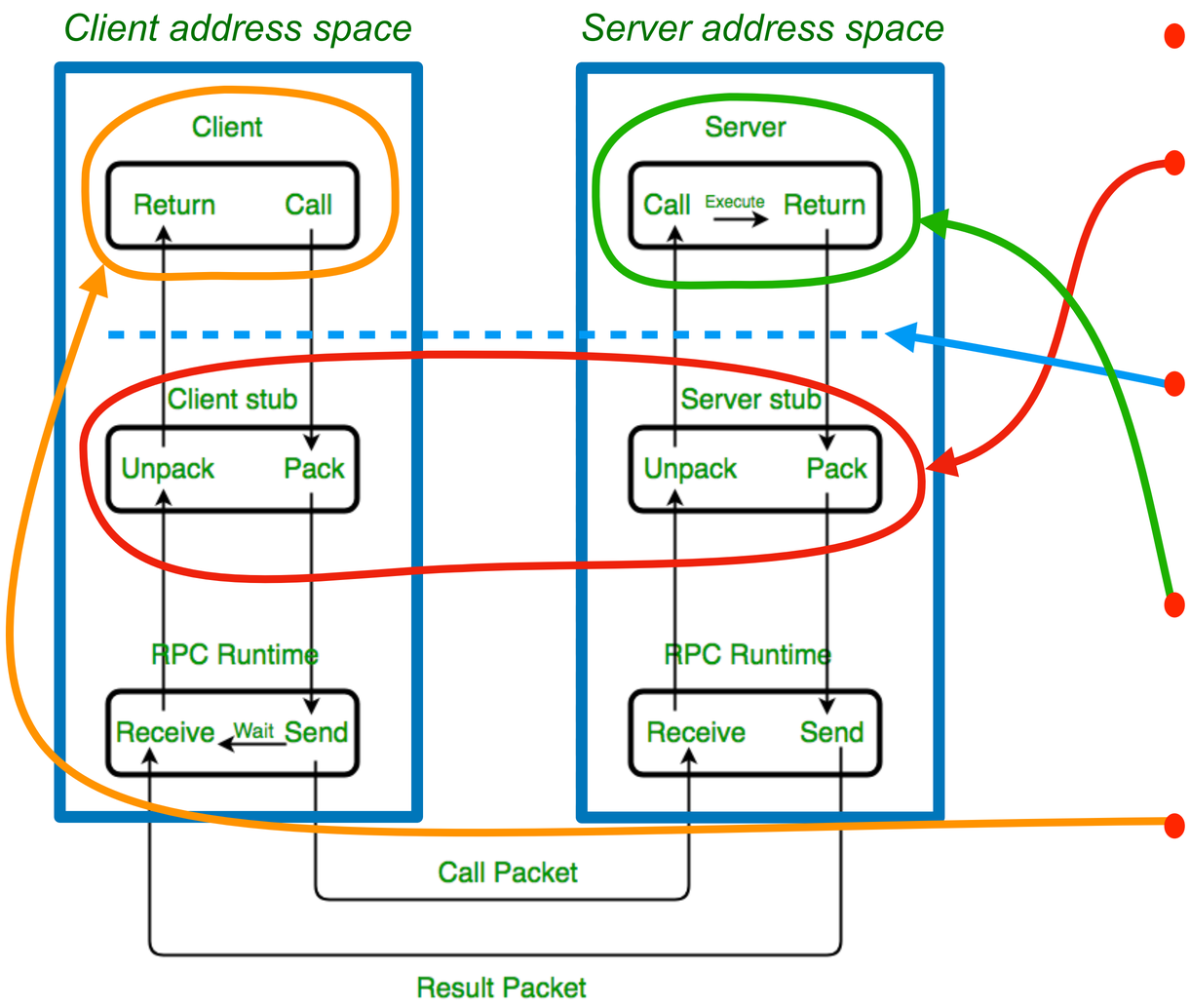
- Define the service in an IDL file
- Generate message implementations using the IDL compiler
- Generate server and client code using the RPC compiler

Workflow for writing RPC-based systems



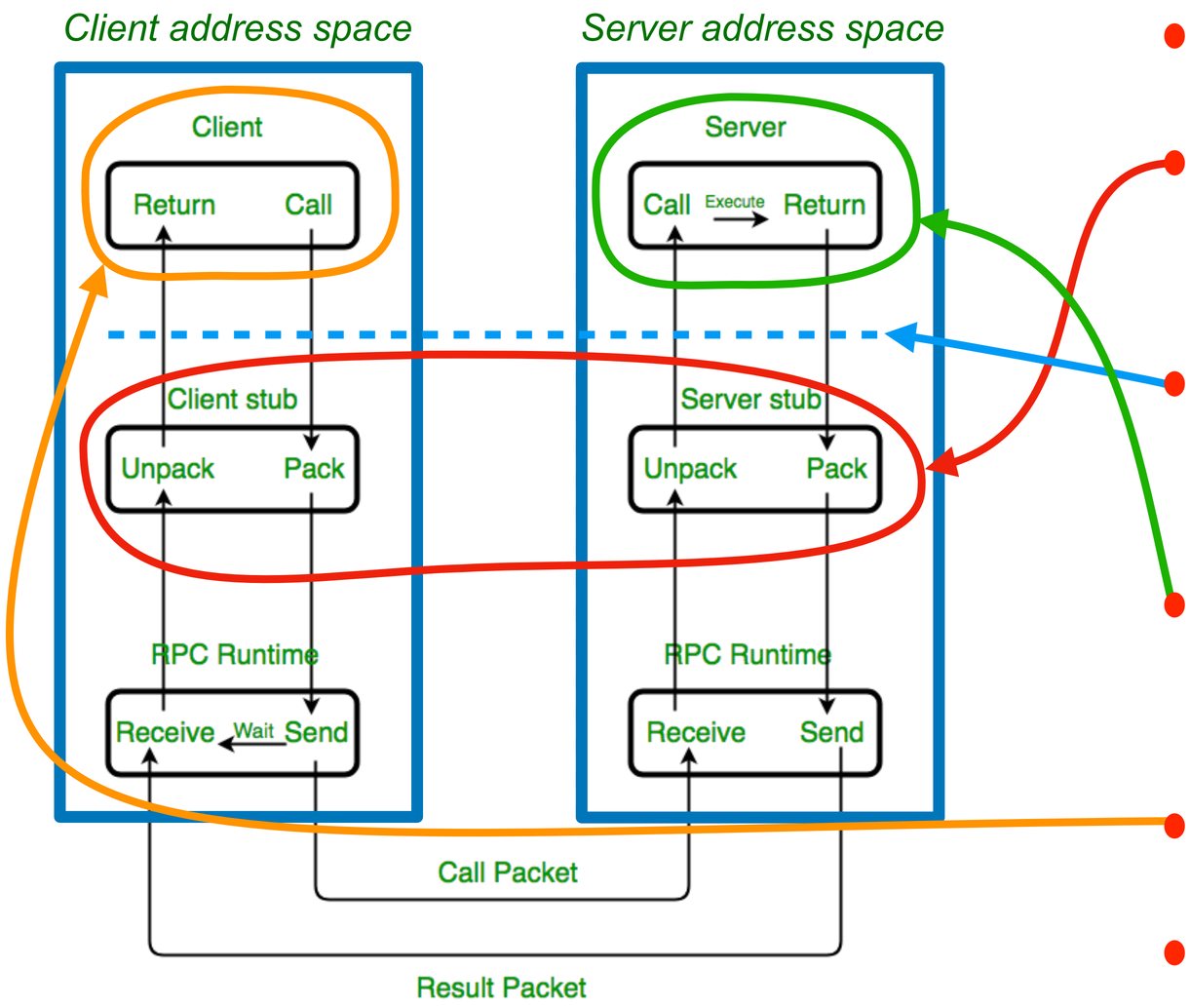
- Define the service in an IDL file
- Generate message implementations using the IDL compiler
- Generate server and client code using the RPC compiler
- Write the server to implement the generated interface

Workflow for writing RPC-based systems



- Define the service in an IDL file
- Generate message implementations using the IDL compiler
- Generate server and client code using the RPC compiler
- Write the server to implement the generated interface
- Write the client to use the interface

Workflow for writing RPC-based systems



- Define the service in an IDL file
- Generate message implementations using the IDL compiler
- Generate server and client code using the RPC compiler
- Write the server to implement the generated interface
- Write the client to use the interface
- Compile, deploy, run

Methods	
`BatchCreateSessions`	Creates multiple new sessions.
`BatchWrite`	Batches the supplied mutation groups in a collection of efficient transactions.
`BeginTransaction`	Begins a new transaction.
`Commit`	Commits a transaction.
`CreateSession`	Creates a new session.
`DeleteSession`	Ends a session, releasing server resources associated with it.
`ExecuteBatchDml`	Executes a batch of SQL DML statements.
`ExecuteSql`	Executes an SQL statement, returning all results in a single reply.
`ExecuteStreamingSql`	Like `ExecuteSql`, except returns the result set as a stream.
`GetSession`	Gets a session.
`ListSessions`	Lists all sessions in a given database.
`PartitionQuery`	Creates a set of partition tokens that can be used to execute a query operation in parallel.
`PartitionRead`	Creates a set of partition tokens that can be used to execute a read operation in parallel.
`Read`	Reads rows from the database using key lookups and scans, as a simple key/value style alternative to `ExecuteSql`.
`Rollback`	Rolls back a transaction, releasing any locks it holds.
`StreamingRead`	Like `Read`, except returns the result set as a stream.

Methods	
BatchCreateSessions	Creates multiple new sessions.
BatchWrite	Batches the supplied mutation groups in a collection of efficient transactions.
BeginTransaction	Begins a new t
Commit	Commits a tra
CreateSession	Creates a new
DeleteSession	Ends a sessio
ExecuteBatchDml	Executes a bat
ExecuteSql	Executes an SQL statement, returning all results in a single reply.
ExecuteStreamingSql	Like ExecuteSql, except returns the result set as a stream.
GetSession	Gets a session.
ListSessions	Lists all sessions in a given database.
PartitionQuery	Creates a set of partition tokens that can be used to execute a query operation in parallel.
PartitionRead	Creates a set of partition tokens that can be used to execute a read operation in parallel.
Read	Reads rows from the database using key lookups and scans, as a simple key/value style alternative to ExecuteSql.
Rollback	Rolls back a transaction, releasing any locks it holds.
StreamingRead	Like Read, except returns the result set as a stream.

rpc ExecuteSql(ExecuteSqlRequest) returns (ResultSet)

Executes an SQL statement, returning all results in a single reply. This method cannot be used to return a result set larger than 10 MiB; if the query yields more data than that, the query fails with a **FAILED_PRECONDITION** error.

Operations inside read-write transactions might return **ABORTED**. If this occurs, the application should restart the transaction from the beginning. See **Transaction** for more details.

Larger result sets can be fetched in streaming fashion by calling **ExecuteStreamingSql** instead.

Methods	
BatchCreateSessions	Creates multiple new sessions.
BatchWrite	Batches the supplied mutation groups in a collection of efficient transactions.
BeginTransaction	Begins a new t
Commit	Commits a tra
CreateSession	Creates a new
DeleteSession	Ends a sessio
ExecuteBatchDml	Executes a bat
ExecuteSql	Executes an SQL statement, returning all results in a single reply.
ExecuteStreamingSql	Like ExecuteSql, except returns the result set as a stream.
GetSession	Gets a session.
ListSessions	Lists all sessions in a given database.
PartitionQuery	Creates a set of partition tokens that can be used to e
PartitionRead	Creates a set of partition tokens that can be used to e
Read	Reads rows from the database using key lookups and to ExecuteSql.
Rollback	Rolls back a transaction, releasing any locks it holds.
StreamingRead	Like Read, except returns the result set as a stream.

rpc ExecuteSql(ExecuteSqlRequest) returns (ResultSet)

Executes an SQL statement, returning all results in a single reply. This method cannot be used to return a result set larger than 10 MiB; if the query yields more data than that, the query fails with a **FAILED_PRECONDITION** error.

Operations inside read-write transactions might return **ABORTED**. If this occurs, the application should restart the transaction from the beginning. See **Transaction** for more details.

Larger result sets can be fetched in streaming fashion by calling **ExecuteStreamingSql** instead.

```
void QueryDataWithStruct(google::cloud::spanner::Client client) {
  namespace spanner = ::google::cloud::spanner;
  using NameType = std::tuple<std::string, std::string>;
  auto singer_info = NameType{"Elena", "Campbell"};
  auto rows = client.ExecuteQuery(spanner::SqlStatement(
      "SELECT SingerId FROM Singers WHERE (FirstName, LastName) = @name",
      {{"name", spanner::Value(singer_info)}}));

  for (auto& row : spanner::StreamOf<std::tuple<std::int64_t>>(rows)) {
    if (!row) throw std::move(row).status();
    std::cout << "SingerId: " << std::get<0>(*row) << "\n";
  }
}
```

Benefits of RPC

- Strong modularity with the convenience of a procedure call
- Reduce fate sharing by exposing callee failures in a controlled manner
  - *This means the caller can now recover easily (esp. if asynchronous RPC)*

Drawbacks of RPC

- RPCs typically take longer than a local procedure call
  - *Leaky abstraction*
- Issues of trust
  - *How do I know who is making the request?*
  - *How do I know the message was not tampered with?*
  - *… ?*
- What does “no response” imply?

RPC Semantics

- At-least-once semantics ≥1 execution
- At-most-once semantics ≤1 executions
- Exactly-once semantics =1 execution

How to implement exactly-once RPC semantics ?

RPC Tax in Data Centers

- Network latency and transmission overhead
- Context switching on both client and server
- Memory and serialization overheads
- Network load
- Security and authentication
- Error handling and retries
- ...

Recap

same address space

- Local procedure calls (module = procedure)
- Program objects & types (module = memory objects)

Memory safety

- Client/server architecture (different address spaces)
- *Example: RPC*

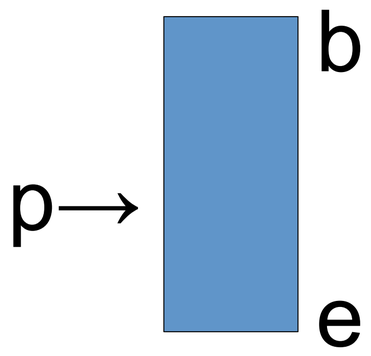
Message-based communication

separate address spaces

THE END

Memory Safety

- Memory can be defined (allocated) or undefined (not allocated)
  - *Deallocated memory cannot be reused prior to (re)allocation*
- Pointer is a capability `(p,b,e)`
  - *Base **b**, extent **e**, pointer **p***



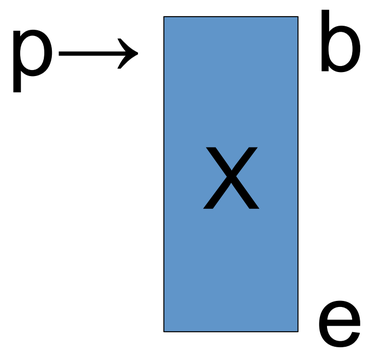
- `*p` is safe iff it accesses memory within the target obj that `p` is based on
- An execution is memory-safe <=> all ptr derefs in that exec are safe
- A program is memory-safe <=> all possible executions (for all possible inputs) are memory-safe

Based on Nagarakatte et al., *SoftBound: Highly Compatible and Complete Spatial Memory Safety for C*, PLDI 2009

"Based on" relationship

- p is based on memory object X iff p is

1. *obtained by allocating* X *at runtime on the heap, or*
2. *obtained as* &X *where* X *is statically allocated, or*
   - e.g., local or global variable, control flow target
3. *obtained as* `&X.foo` *(i.e., from field of* X*), or*
4. *the result of a computation involving operands that are ptrs based on* X *or non-ptrs*
   - copy of another pointer
   - valid pointer arithmetic
   - array indexing



Memory Safety

- Pointer is a capability `(p,b,e)`
  - *Base **b**, extent **e**, pointer **p***
- `*p` is safe iff accesses memory within the target obj that `p` is based on[1]

```
b <= p <= e
```

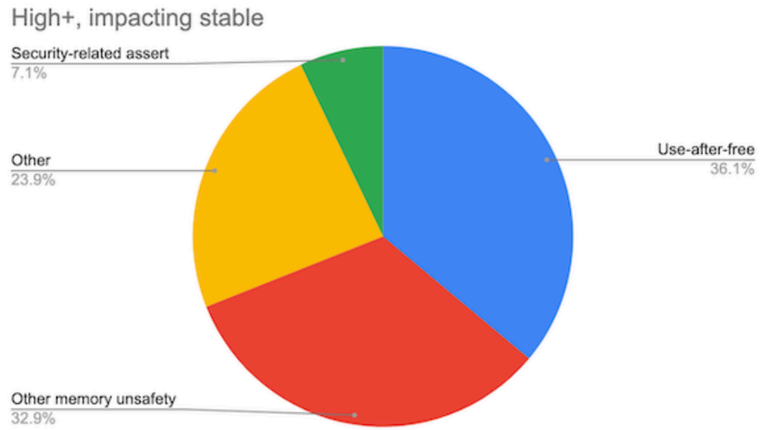
- An execution is memory-safe <=>
  all pointer dereferences in that execution are safe
- A program is memory-safe <=>
  all possible executions (for all possible inputs) are memory-safe

[1] and that memory is defined

Memory Safety

- Memory safety is fundamental to in-memory client/server
- A pointer is a name for X => set of names for reaching X is transitive closure over "based-on" relationship
- Spatial vs. temporal violations of memory safety

Around 70% of our high severity security bugs are memory unsafety problems (that is, mistakes with C/C++ pointers). Half of those are use-after-free bugs.



https://www.chromium.org/Home/chromium-security/memory-

Google Protobuf & gRPC

Example of how to write code that uses RPC

Interface Definition Language (Google protobuf)

```
message Person {
  required string name = 1;
  required int32 id = 2;
  optional string email = 3;

  enum PhoneType {
    MOBILE = 0;
    HOME = 1;
    WORK = 2;
  }

  message PhoneNumber {
    required string number = 1;
    optional PhoneType type = 2 [default = HOME];
  }

  repeated PhoneNumber phones = 4;
}

message AddressBook {
  repeated Person people = 1;
}
```

contacts.proto

Interface Definition Language (Google protobuf)

```
message Person {
  required string name = 1;
  required int32 id = 2;
  optional string email = 3;

  enum PhoneType {
    MOBILE = 0;
    HOME = 1;
    WORK = 2;
  }

  message PhoneNumber {
    required string number = 1;
    optional PhoneType type = 2 [default = HOME];
  }

  repeated PhoneNumber phones = 4;
}

message AddressBook {
  repeated Person people = 1;
}
```

contacts.proto

Interface Definition Language (Google protobuf)

```
message Person {
  required string name = 1;
  required int32 id = 2;
  optional string email = 3;

  enum PhoneType {
    MOBILE = 0;
    HOME = 1;
    WORK = 2;
  }

  message PhoneNumber {
    required string number = 1;
    optional PhoneType type = 2 [default = HOME];
  }

  repeated PhoneNumber phones = 4;
}

message AddressBook {
  repeated Person people = 1;
}
```

contacts.proto

Interface Definition Language (Google protobuf)

```
message Person {
  required string name = 1;
  required int32 id = 2;
  optional string email = 3;

  enum PhoneType {
    MOBILE = 0;
    HOME = 1;
    WORK = 2;
  }

  message PhoneNumber {
    required string number = 1;
    optional PhoneType type = 2 [default = HOME];
  }

  repeated PhoneNumber phones = 4;
}

message AddressBook {
  repeated Person people = 1;
}
```

contacts.proto

Interface Definition Language (Google protobuf)

```
message Person {
  required string name = 1;
  required int32 id = 2;
  optional string email = 3;

  enum PhoneType {
    MOBILE = 0;
    HOME = 1;
    WORK = 2;
  }

  message PhoneNumber {
    required string number = 1;
    optional PhoneType type = 2 [default = HOME];
  }

  repeated PhoneNumber phones = 4;
}

message AddressBook {
  repeated Person people = 1;
}
```

contacts.proto

Interface Definition Language (Google protobuf)

```
message Person {
  required string name = 1;
  required int32 id = 2;
  optional string email = 3;

  enum PhoneType {
    MOBILE = 0;
    HOME = 1;
    WORK = 2;
  }

  message PhoneNumber {
    required string number = 1;
    optional PhoneType type = 2 [default = HOME];
  }

  repeated PhoneNumber phones = 4;
}

message AddressBook {
  repeated Person people = 1;
}
```

contacts.proto

Interface Definition Language (Google protobuf)

```
message Person {
  required string name = 1;
  required int32 id = 2;
  optional string email = 3;

  enum PhoneType {
    MOBILE = 0;
    HOME = 1;
    WORK = 2;
  }

  message PhoneNumber {
    required string number = 1;
    optional PhoneType type = 2 [default = HOME];
  }

  repeated PhoneNumber phones = 4;
}

message AddressBook {
  repeated Person people = 1;
}
```

contacts.proto

```
protoc --cpp_out=$DST_DIR contacts.proto
```

contacts.pb.h
contacts.pb.cc

contacts.pb.h

```
// name
inline bool has_name() const;
inline void clear_name();
inline const ::std::string& name() const;
inline void set_name(const ::std::string& value);
inline void set_name(const char* value);
inline ::std::string* mutable_name();

// id
inline bool has_id() const;
inline void clear_id();
inline int32_t id() const;
inline void set_id(int32_t value);

// email
inline bool has_email() const;
inline void clear_email();
inline const ::std::string& email() const;
inline void set_email(const ::std::string& value);
inline void set_email(const char* value);
inline ::std::string* mutable_email();

// phones
inline int phones_size() const;
inline void clear_phones();
inline const ::google::protobuf::RepeatedPtrField< ::pocs::Person_PhoneNumber >& phones() const;
inline ::google::protobuf::RepeatedPtrField< ::pocs::Person_PhoneNumber >* mutable_phones();
inline const ::Person_PhoneNumber& phones(int index) const;
inline ::Person_PhoneNumber* mutable_phones(int index);
inline ::Person_PhoneNumber* add_phones();
```

Interface Definition Language (Google protobuf)

```
message Person {
  required string name = 1;
  required int32 id = 2;
  optional string email = 3;

  enum PhoneType {
    MOBILE = 0;
    HOME = 1;
    WORK = 2;
  }

  message PhoneNumber {
    required string number = 1;
    optional PhoneType type = 2 [default = HOME];
  }

  repeated PhoneNumber phones = 4;
}

message AddressBook {
  repeated Person people = 1;
}
```

contacts.proto


```
// serializes the message and stores the bytes in the given string.
// The bytes are binary, not text; we only use the string class as
// a convenient container.
bool SerializeToString(string* output) const;

// parses a message from the given string.
bool ParseFromString(const string& data);

// writes the message to the given C++ ostream.
bool SerializeToOstream(ostream* output) const;

// parses a message from the given C++ istream.
bool ParseFromIstream(istream* input);
```

RPC stubs (gRPC)

```
// Interface exported by the server.
service Contacts {
  // A simple RPC.
  //
  // Obtains the feature of a given Person.
  rpc GetNumber(Person) returns (PhoneNumber) {}

  // A server-to-client streaming RPC.
  //
  // Obtains the PhoneNumbers available for the given Person.
  rpc ListNumbers(Person) returns (stream PhoneNumber) {}

  ...
}
message Person ...
```

contacts.proto

RPC stubs (gRPC)

```
// Interface exported by the server.
service Contacts {
  // A simple RPC.
  //
  // Obtains the feature of a given Person.
  rpc GetNumber(Person) returns (PhoneNumber) {}

  // A server-to-client streaming RPC.
  //
  // Obtains the PhoneNumbers available for the given Person.
  rpc ListNumbers(Person) returns (stream PhoneNumber) {}

  ...
}
message Person ...
```

contacts.proto


```
protoc --grpc_out=. --plugin=protoc-gen-grpc=$PLUGIN_DIR contacts.proto
```



```
contacts.grpc.pb.h
contacts.grpc.pb.cc
```

- remote interface type (“stub”) for clients
- abstract interface for servers to implement

REST vs. RPC

REST vs. RPC

REST vs. RPC

- = Representational State Transfer
  - *REST imposes a resource-oriented thinking, in contrast to RPC (action-oriented)*
  - *CRUD, and the set of legal actions from any state is always controlled by the server*

REST vs. RPC

- = Representational State Transfer
  - *REST has a resource-oriented thinking, while RPC is action-oriented*
  - *CRUD, and the set of legal actions from any state is always controlled by the server*
- All communication is stateless server-side and cacheable

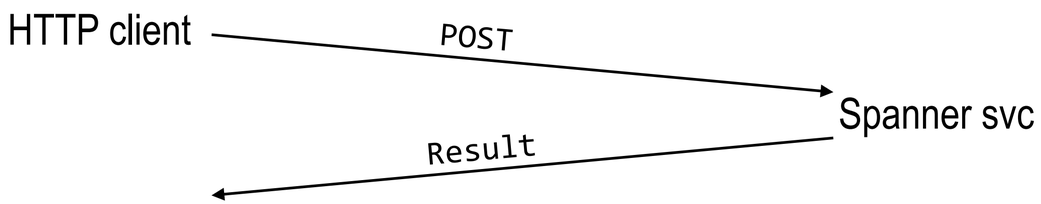
REST vs. RPC

- = Representational State Transfer
  - *REST has a resource-oriented thinking, while RPC is action-oriented*
  - *CRUD, and the set of legal actions from any state is always controlled by the server*
- All communication is stateless server-side and cacheable
- Most popular data representation = JSON

REST vs. RPC

- = Representational State Transfer
  - *REST has a resource-oriented thinking, while RPC is action-oriented*
  - *CRUD, and the set of legal actions from any state is always controlled by the server*
- All communication is stateless server-side and cacheable
- Most popular data representation = JSON
- REST is often (~always) done over HTTP
  - *GET, POST/PUT or DELETE requests*
  - *avoid reinventing the wheel (e.g., metadata for caching)*

Methods	
batchCreate	POST /v1/{database=projects/*/instances/*/databases/*}/sessions:batchCreate Creates multiple new sessions.
batchWrite	POST /v1/{session=projects/*/instances/*/databases/*/sessions/*}:batchWrite Batches the supplied mutation groups in a collection of efficient transactions.
beginTransaction	POST /v1/{session=projects/*/instances/*/databases/*/sessions/*}:beginTransaction Begins a new transaction.
commit	POST /v1/{session=projects/*/instances/*/databases/*/sessions/*}:commit Commits a transaction.
create	POST /v1/{database=projects/*/instances/*/databases/*}/sessions Creates a new session.
delete	DELETE /v1/{name=projects/*/instances/*/databases/*/sessions/*} Ends a session, releasing server resources associated with it.
executeBatchDml	POST /v1/{session=projects/*/instances/*/databases/*/sessions/*}:executeBatchDml Executes a batch of SQL DML statements.
executeSql	POST /v1/{session=projects/*/instances/*/databases/*/sessions/*}:executeSql Executes an SQL statement, returning all results in a single reply.
executeStreamingSql	POST /v1/{session=projects/*/instances/*/databases/*/sessions/*}:executeStreamingSql Like ExecuteSql, except returns the result set as a stream.
get	GET /v1/{name=projects/*/instances/*/databases/*/sessions/*} Gets a session.
list	GET /v1/{database=projects/*/instances/*/databases/*}/sessions Lists all sessions in a given database.
partitionQuery	POST /v1/{session=projects/*/instances/*/databases/*/sessions/*}:partitionQuery Creates a set of partition tokens that can be used to execute a query operation in parallel.
partitionRead	POST /v1/{session=projects/*/instances/*/databases/*/sessions/*}:partitionRead Creates a set of partition tokens that can be used to execute a read operation in parallel.
read	POST /v1/{session=projects/*/instances/*/databases/*/sessions/*}:read Reads rows from the database using key lookups and scans, as a simple key/value style alternative to ExecuteSql.
rollback	POST /v1/{session=projects/*/instances/*/databases/*/sessions/*}:rollback Rolls back a transaction, releasing any locks it holds.
streamingRead	POST /v1/{session=projects/*/instances/*/databases/*/sessions/*}:streamingRead Like Read, except returns the result set as a stream.



Methods	
batchCreate	POST /v1/{database=projects/*/instances/*/databases/*}/sessions:batchCreate Creates multiple new sessions.
batchWrite	POST /v1/{session=projects/*/instances/*/databases/*/sessions/*}:batchWrite Batches the supplied mutation groups in a collection of efficient transactions.
beginTransaction	POST /v1/{session=projects/*/instances/*/databases/*/sessions/*}:beginTransaction Begins a new transaction.
commit	POST /v1/{session=projects/*/instances/*/databases/*/sessions/*}:commit Commits a transaction.
create	POST /v1/{database=projects/*/instances/*/databases/*}/sessions Creates a new session.
delete	DELETE /v1/{name=projects/*/instances/*/databases/*/sessions/*} Ends a session, releasing server resources associated with it.
executeBatchDml	POST /v1/{session=projects/*/instances/*/databases/*/sessions/*}:executeBatchDml Executes a batch of SQL DML statements.
executeSql	POST /v1/{session=projects/*/instances/*/databases/*/sessions/*}:executeSql Executes an SQL statement, returning all results in a single reply.
executeStreamingSql	POST /v1/{session=projects/*/instances/*/databases/*/sessions/*}:executeStreamingSql Like ExecuteSql, except returns the result set as a stream.
get	GET /v1/{name=projects/*/instances/*/databases/*/sessions/*} Gets a session.
list	GET /v1/{database=projects/*/instances/*/databases/*}/sessions Lists all sessions in a given database.
partitionQuery	POST /v1/{session=projects/*/instances/*/databases/*/sessions/*}:partitionQuery Creates a set of partition tokens that can be used to execute a query operation in parallel.
partitionRead	POST /v1/{session=projects/*/instances/*/databases/*/sessions/*}:partitionRead Creates a set of partition tokens that can be used to execute a read operation in parallel.
read	POST /v1/{session=projects/*/instances/*/databases/*/sessions/*}:read Reads rows from the database using key lookups and scans, as a simple key/value style alternative to ExecuteSql.
rollback	POST /v1/{session=projects/*/instances/*/databases/*/sessions/*}:rollback Rolls back a transaction, releasing any locks it holds.
streamingRead	POST /v1/{session=projects/*/instances/*/databases/*/sessions/*}:streamingRead Like Read, except returns the result set as a stream.

HTTP method

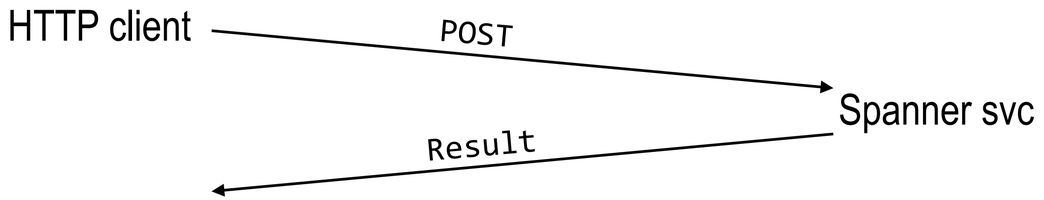
API service

API version

Identifiers for project, instance, database, and session

Endpoint

```
POST https://spanner.googleapis.com/v1/{session=projects/*/instances/*/databases/*/sessions/*}:executeSql
```



Methods	
batchCreate	POST /v1/{database=projects/*/instances/*/databases/*}/sessions:batchCreate Creates multiple new sessions.
batchWrite	POST /v1/{session=projects/*/instances/*/databases/*/sessions/*}:batchWrite Batches the supplied mutation groups in a collection of efficient transactions.
beginTransaction	POST /v1/{session=projects/*/instances/*/databases/*/sessions/*}:beginTransaction Begins a new transaction.
commit	POST /v1/{session=projects/*/instances/*/databases/*/sessions/*}:commit Commits a transaction.
create	POST /v1/{database=projects/*/instances/*/databases/*}/sessions Creates a new session.
delete	DELETE /v1/{name=projects/*/instances/*/databases/*/sessions/*} Ends a session, releasing server resources associated with it.
executeBatchDml	POST /v1/{session=projects/*/instances/*/databases/*/sessions/*}:executeBatchDml Executes a batch of SQL DML statements.
executeSql	POST /v1/{session=projects/*/instances/*/databases/*/sessions/*}:executeSql Executes an SQL statement, returning all results in a single reply.
executeStreamingSql	POST /v1/{session=projects/*/instances/*/databases/*/sessions/*}:executeStreamingSql Like ExecuteSql, except returns the result set as a stream.
get	GET /v1/{name=projects/*/instances/*/databases/*/sessions/*} Gets a session.
list	GET /v1/{database=projects/*/instances/*/databases/*}/sessions Lists all sessions in a given database.
partitionQuery	POST /v1/{session=projects/*/instances/*/databases/*/sessions/*}:partitionQuery Creates a set of partition tokens that can be used to execute a query operation in parallel.
partitionRead	POST /v1/{session=projects/*/instances/*/databases/*/sessions/*}:partitionRead Creates a set of partition tokens that can be used to execute a read operation in parallel.
read	POST /v1/{session=projects/*/instances/*/databases/*/sessions/*}:read Reads rows from the database using key lookups and scans, as a simple key/value style alternative to ExecuteSql.
rollback	POST /v1/{session=projects/*/instances/*/databases/*/sessions/*}:rollback Rolls back a transaction, releasing any locks it holds.
streamingRead	POST /v1/{session=projects/*/instances/*/databases/*/sessions/*}:streamingRead Like Read, except returns the result set as a stream.

HTTP method

API service

API version

Identifiers for project, instance, database, and session

Endpoint

```
POST https://spanner.googleapis.com/v1/{session=projects/*/instances/*/databases/*/sessions/*}:executeSql
```

Request body (JSON)

```
{
  "transaction": {
    object (TransactionSelector)
  },
  "sql": string,
  "params": {
    object
  },
  "paramTypes": {
    string: {
      object (Type)
    },
    ...
  },
  "resumeToken": string,
  "queryMode": enum (QueryMode),
  "partitionToken": string,
  "seqno": string,
  "queryOptions": {
    object (QueryOptions)
  },
  "requestOptions": {
    object (RequestOptions)
  },
  "dataBoostEnabled": boolean
}
```