



# The Client/Server Design Pattern

---

Prof. George Canea

*School of Computer & Communication Sciences*

# Outline

- Local procedure calls (module = procedure) | Memory safety
- Program objects & types (module = memory object)
- Client/server architecture (different address spaces) | Message-based communication
- *Example: RPC*

# Outline

*same address space*

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

Memory safety

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

Message-based communication

# Outline

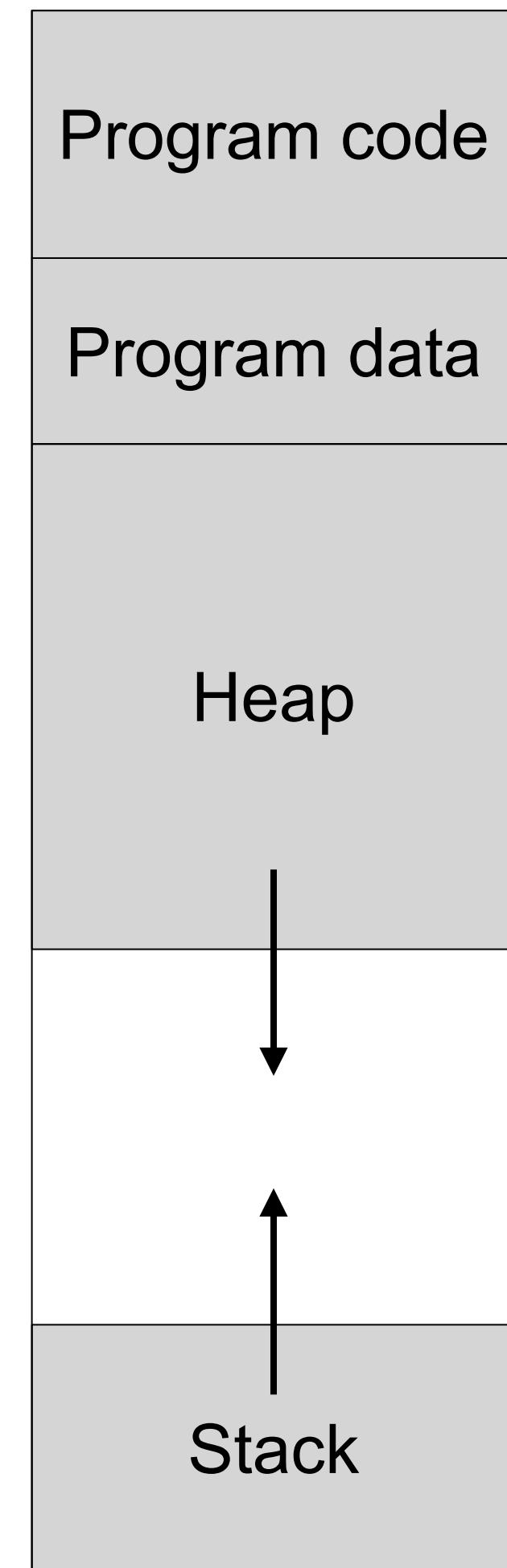
- |   |  | <i>same address space</i>          |
|---|--|------------------------------------|
| • Local procedure calls (module = procedure)            | • Program objects & types (module = memory object) | <b>Memory safety</b>               |
|   |  | <b>Message-based communication</b> |
| <i>separate address spaces</i>                          |  |                                    |
| • Client/server architecture (different address spaces) | • <i>Example: RPC</i>                              | <b>Message-based communication</b> |
|   |  | <b>Memory safety</b>               |

# (Local) Procedure Calls

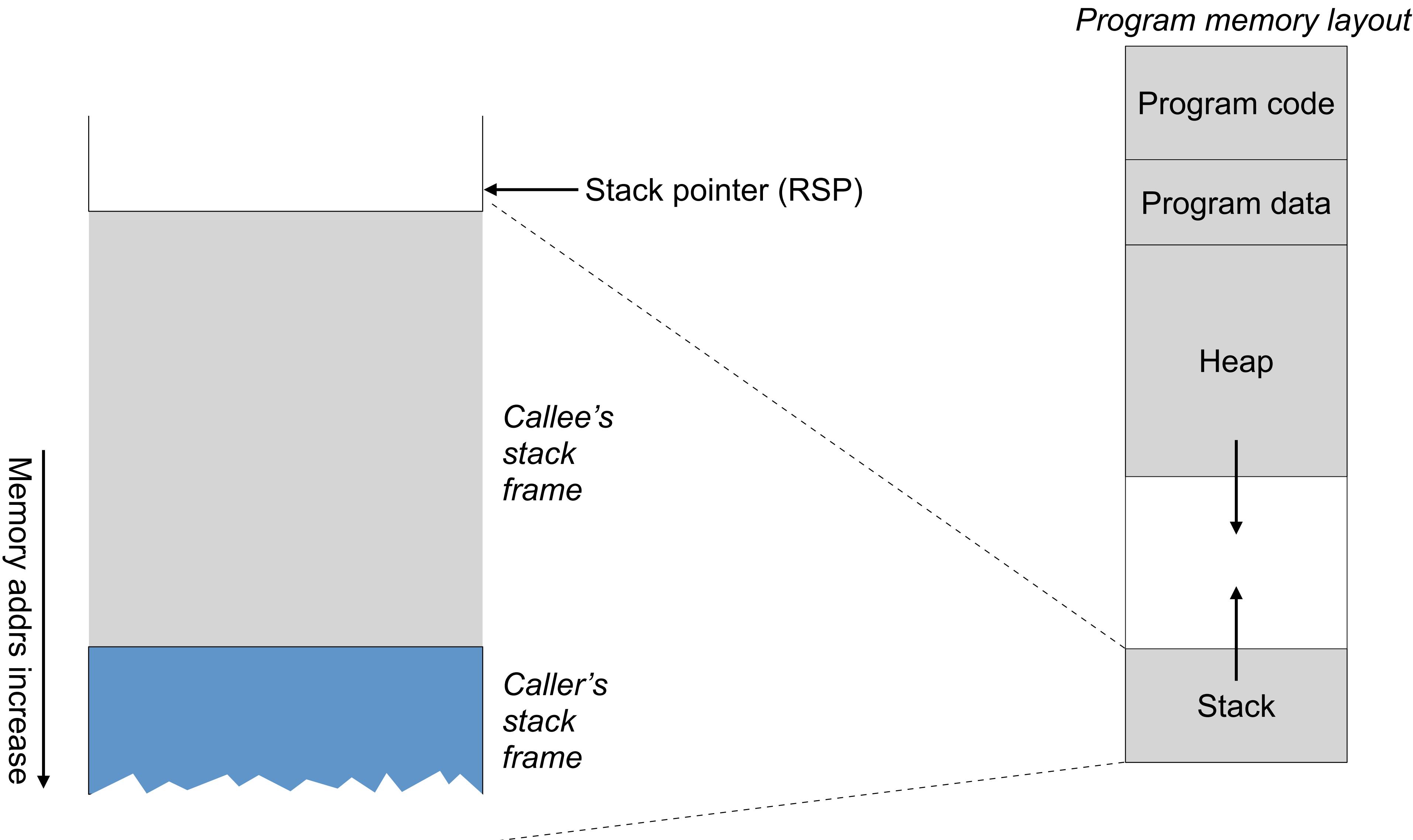
*Basic mechanism for modularizing a program  
(Modules = procedures)*

# Stack-based calling convention

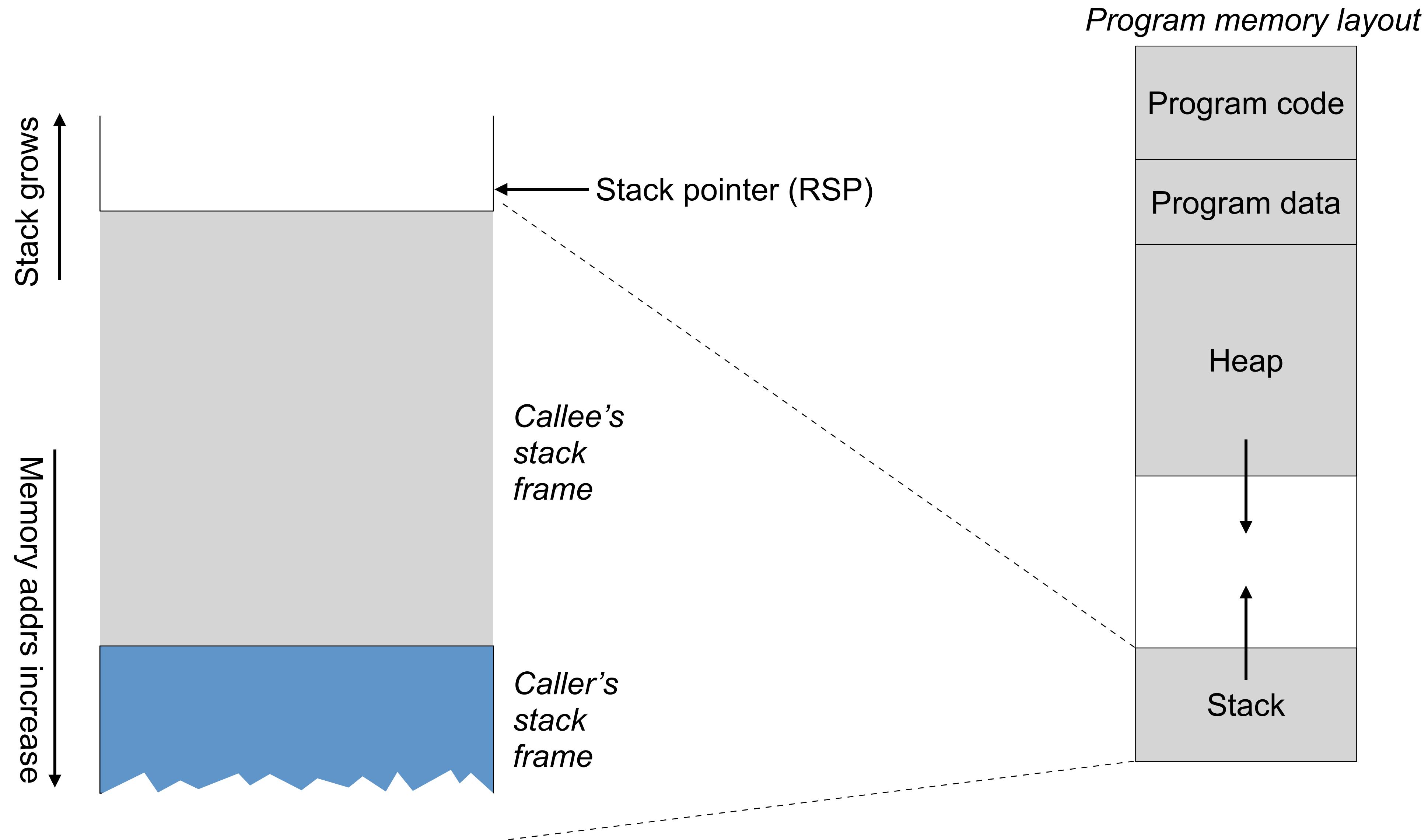
*Program memory layout*



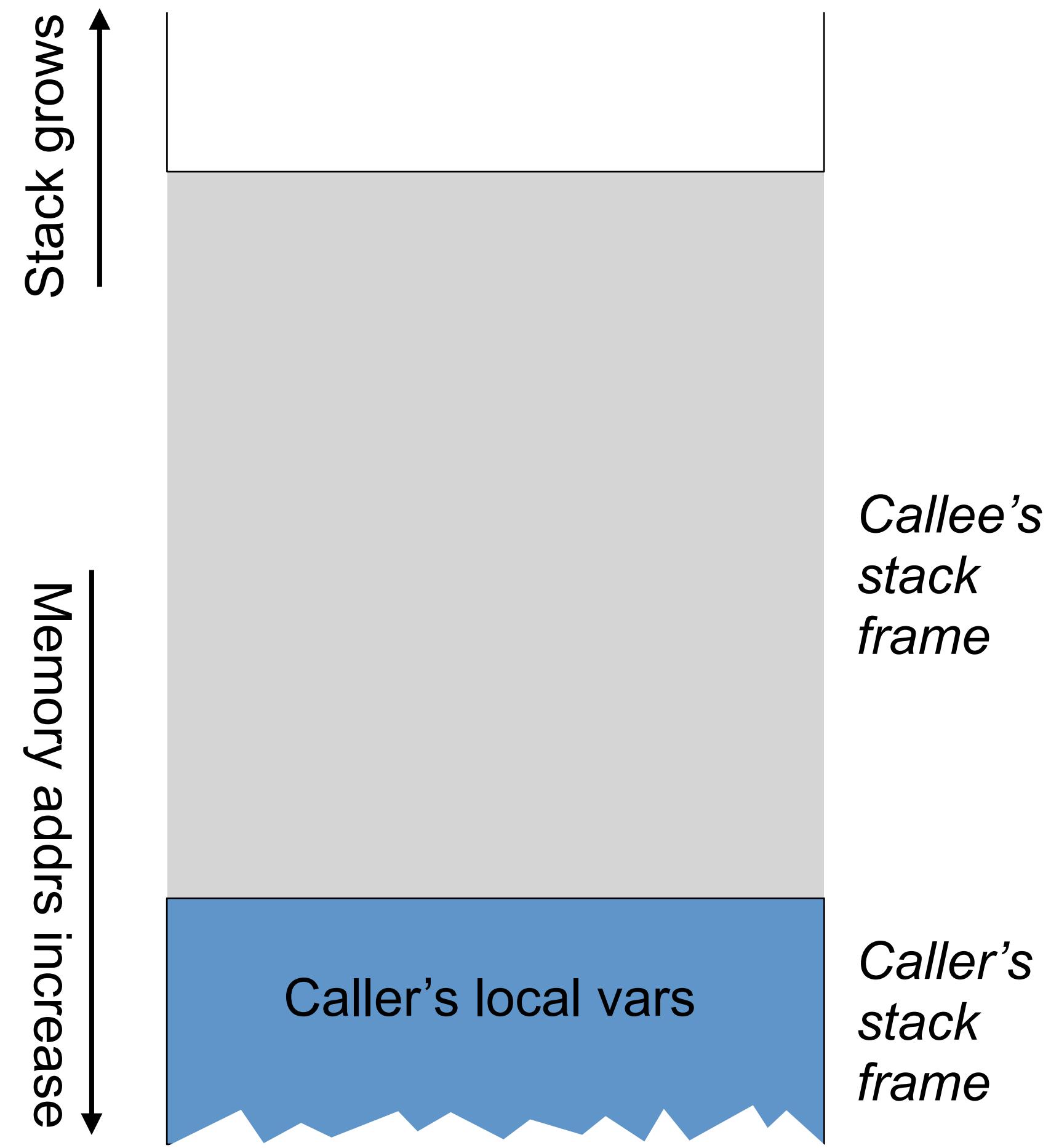
# Stack-based calling convention



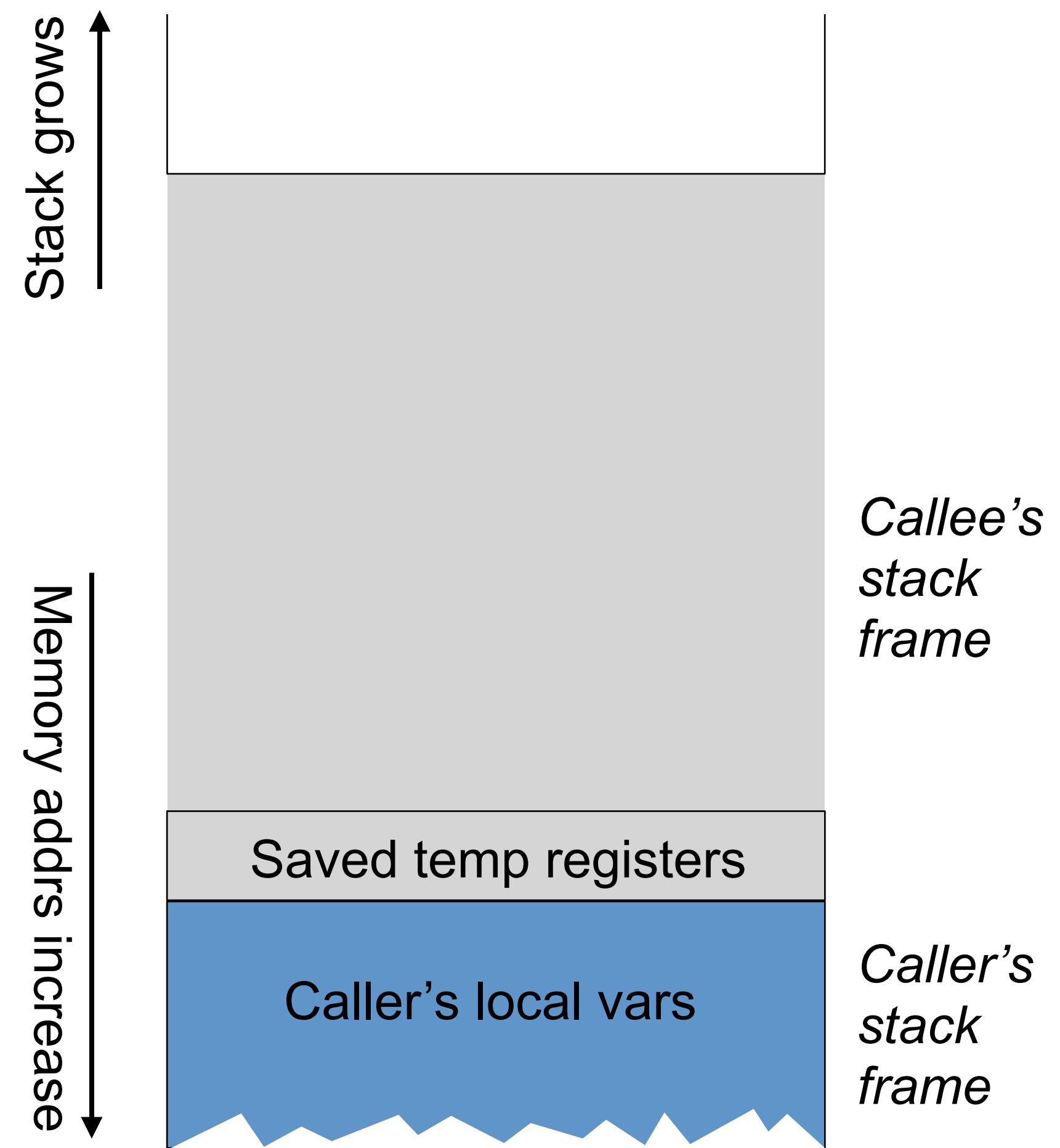
# Stack-based calling convention



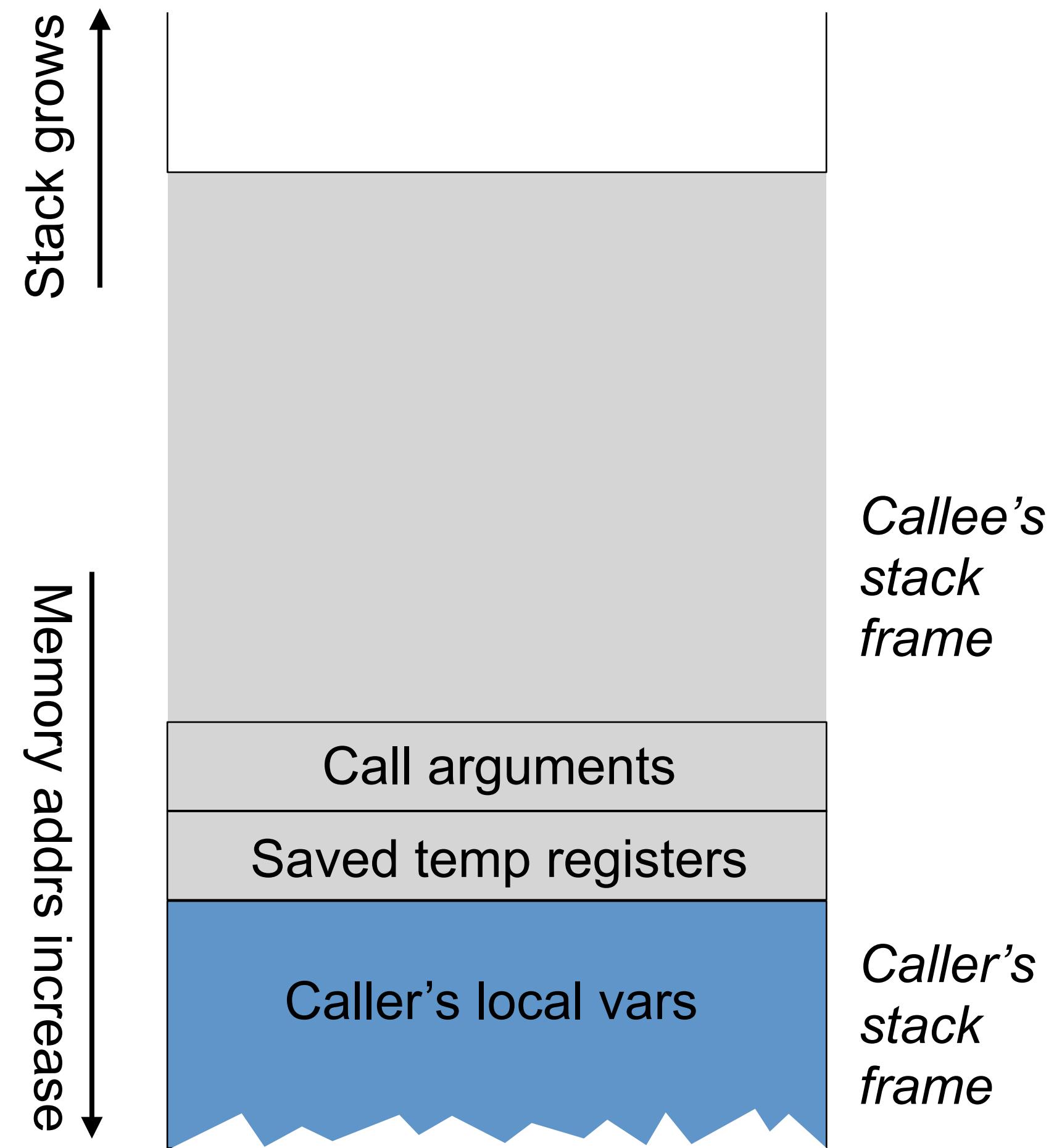
# Stack-based calling convention



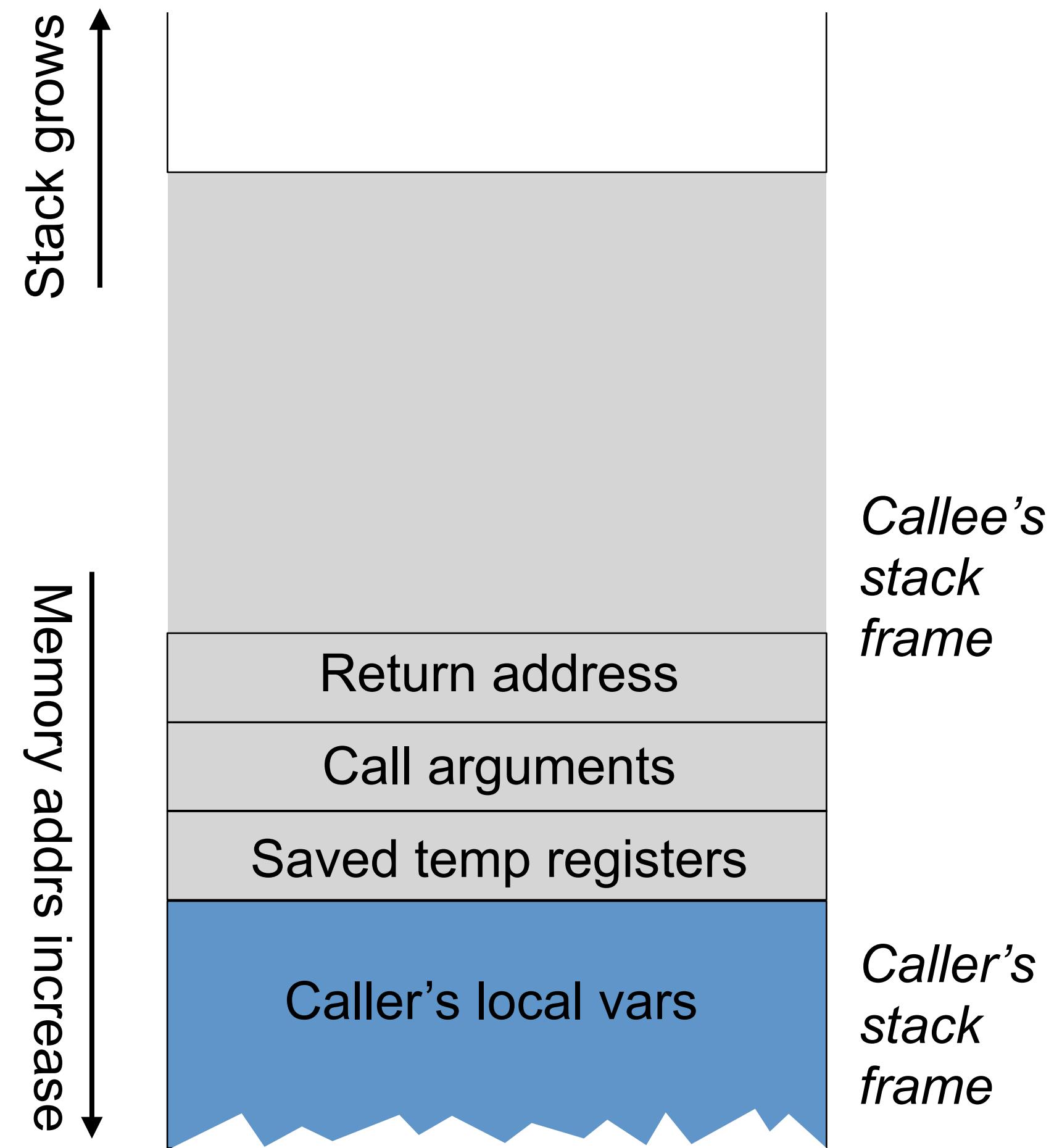
# Stack-based calling convention



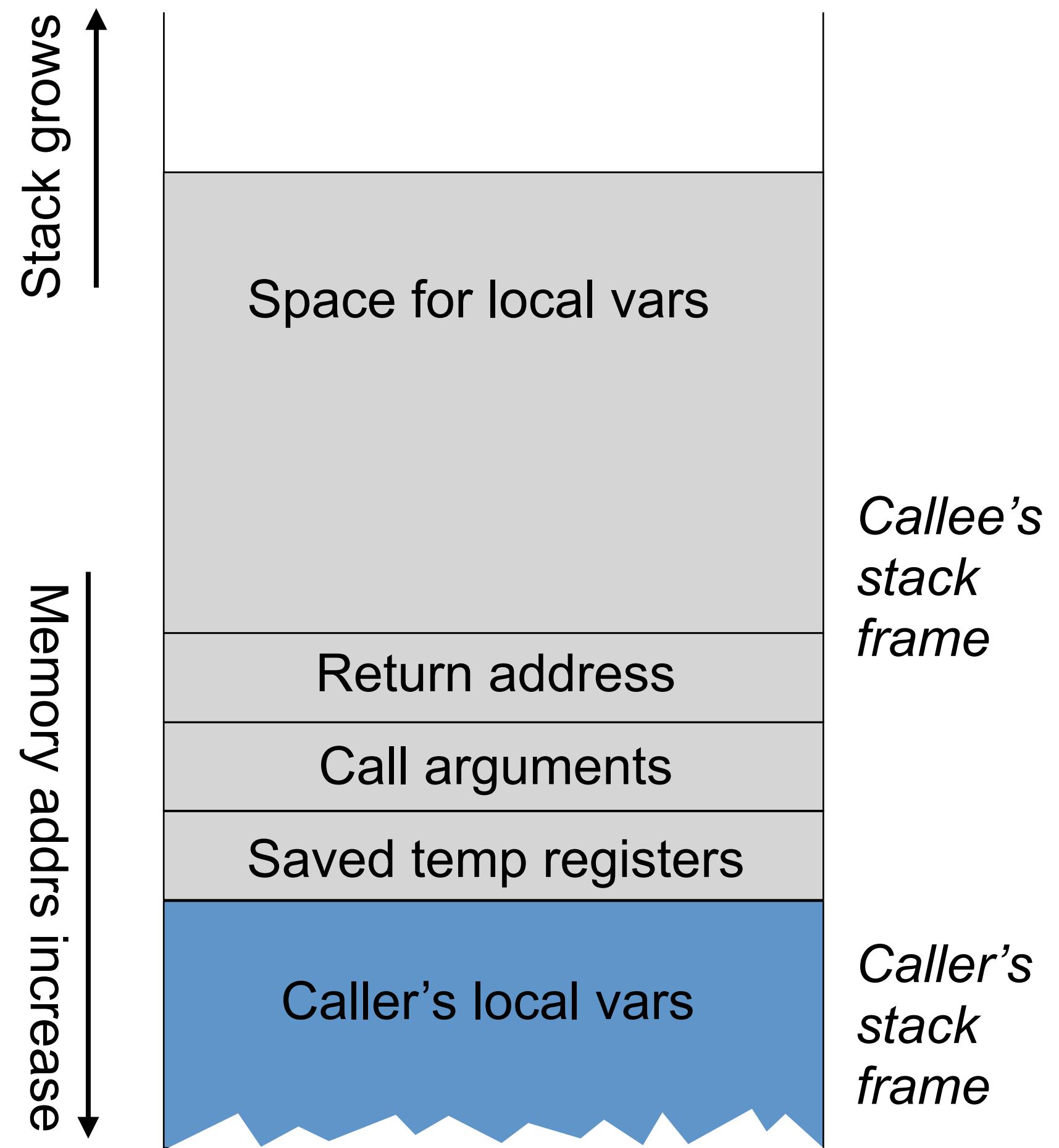
# Stack-based calling convention



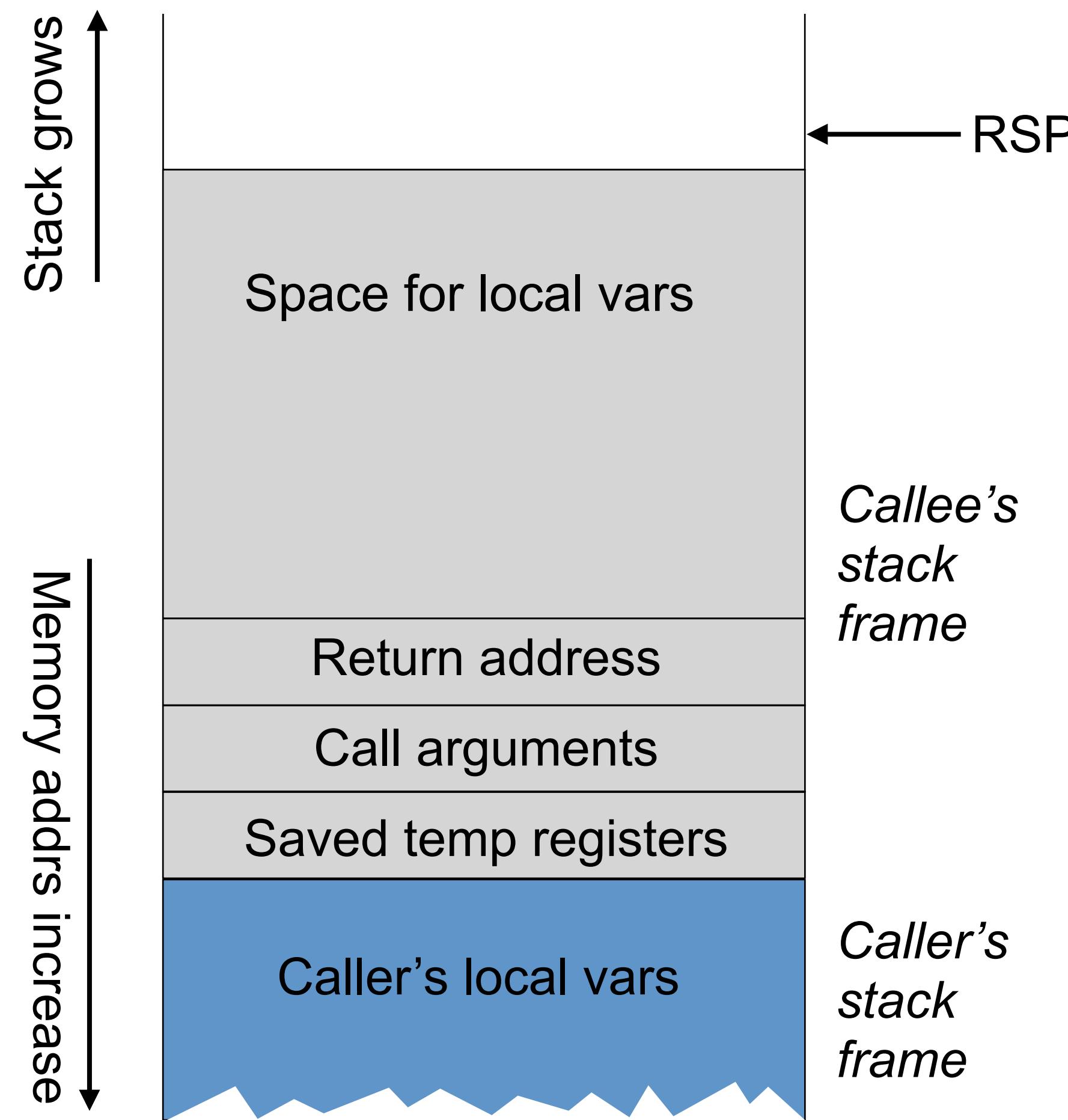
# Stack-based calling convention



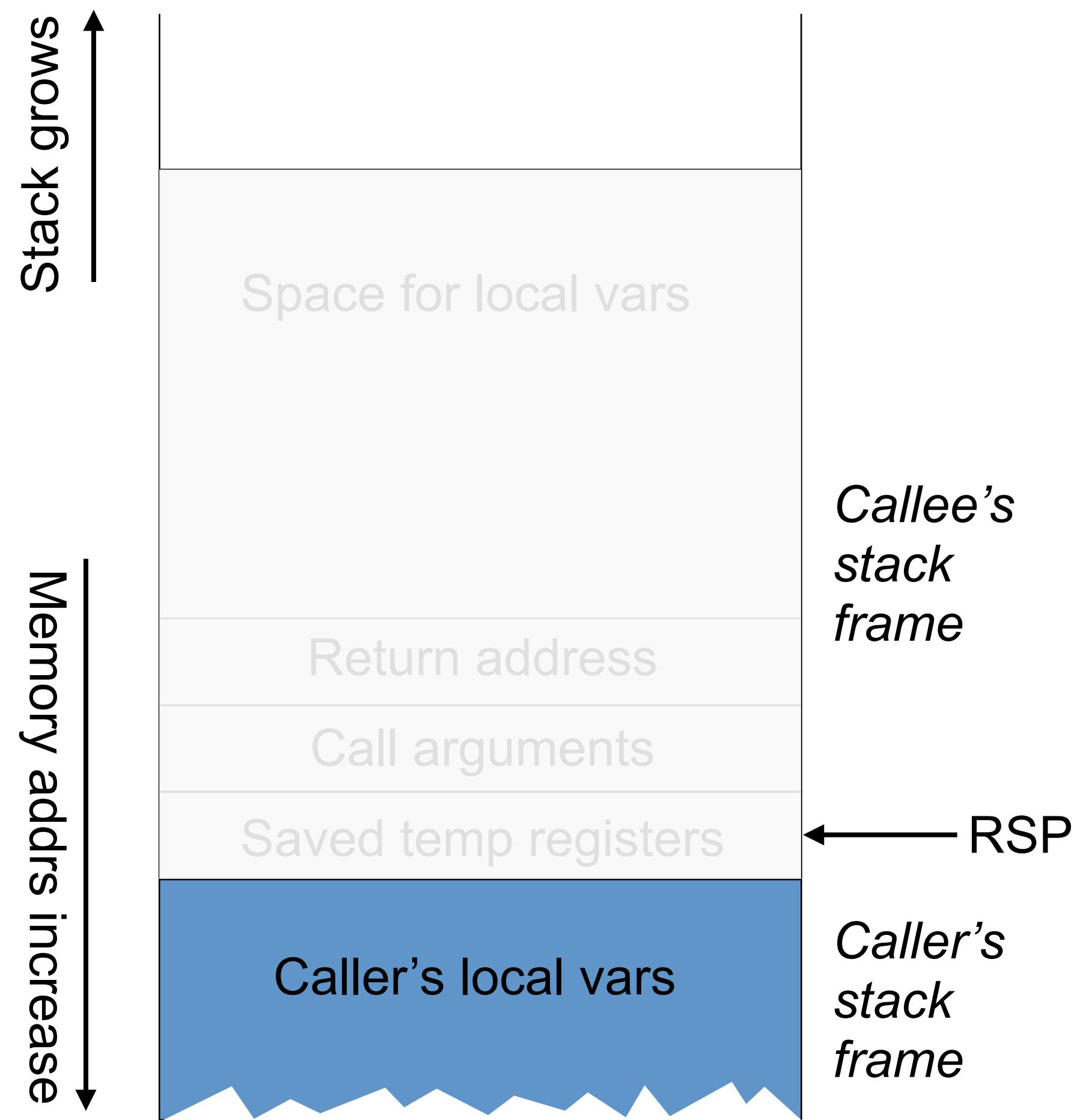
# Stack-based calling convention



# Stack-based calling convention

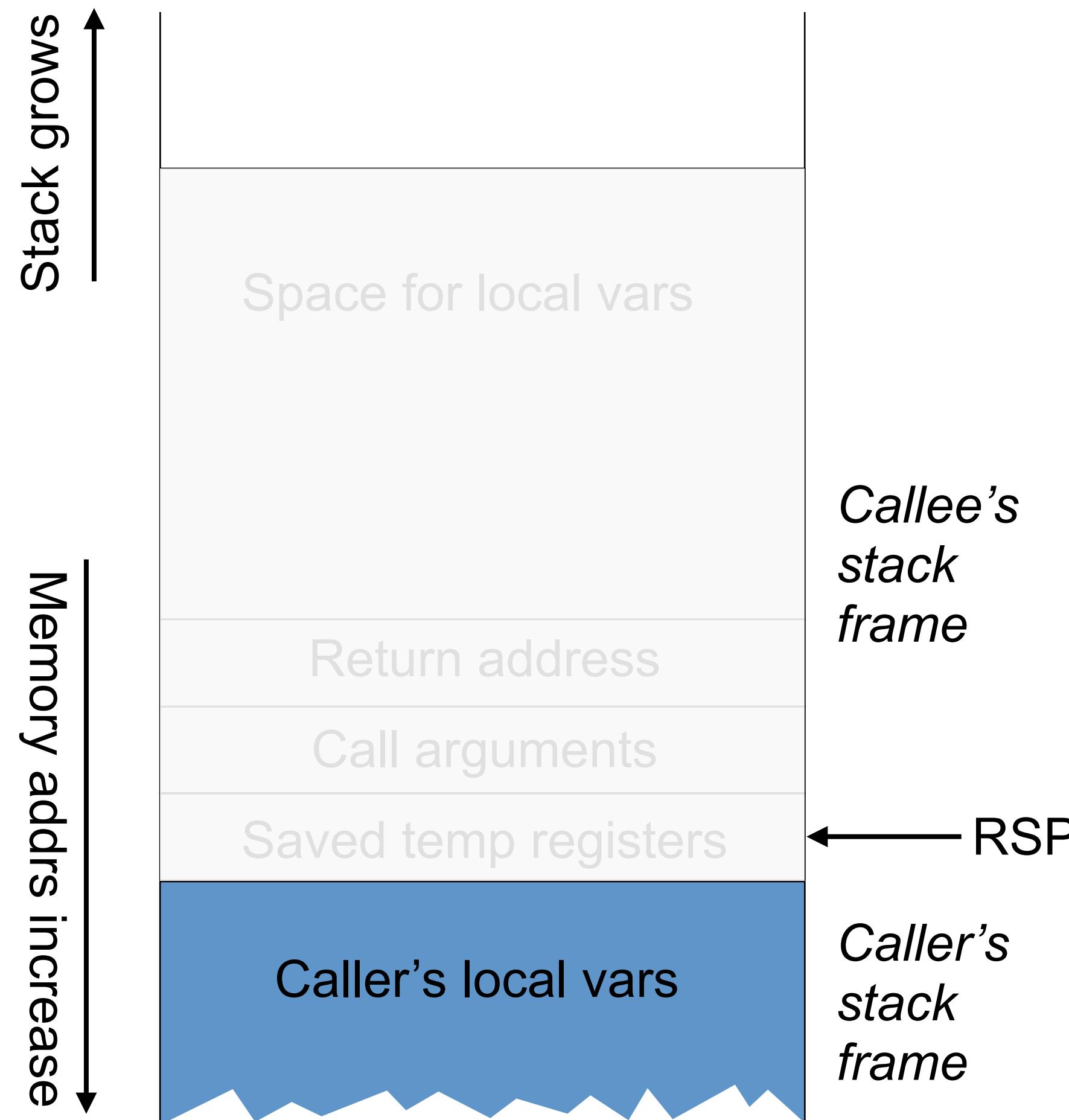


# Stack-based calling convention

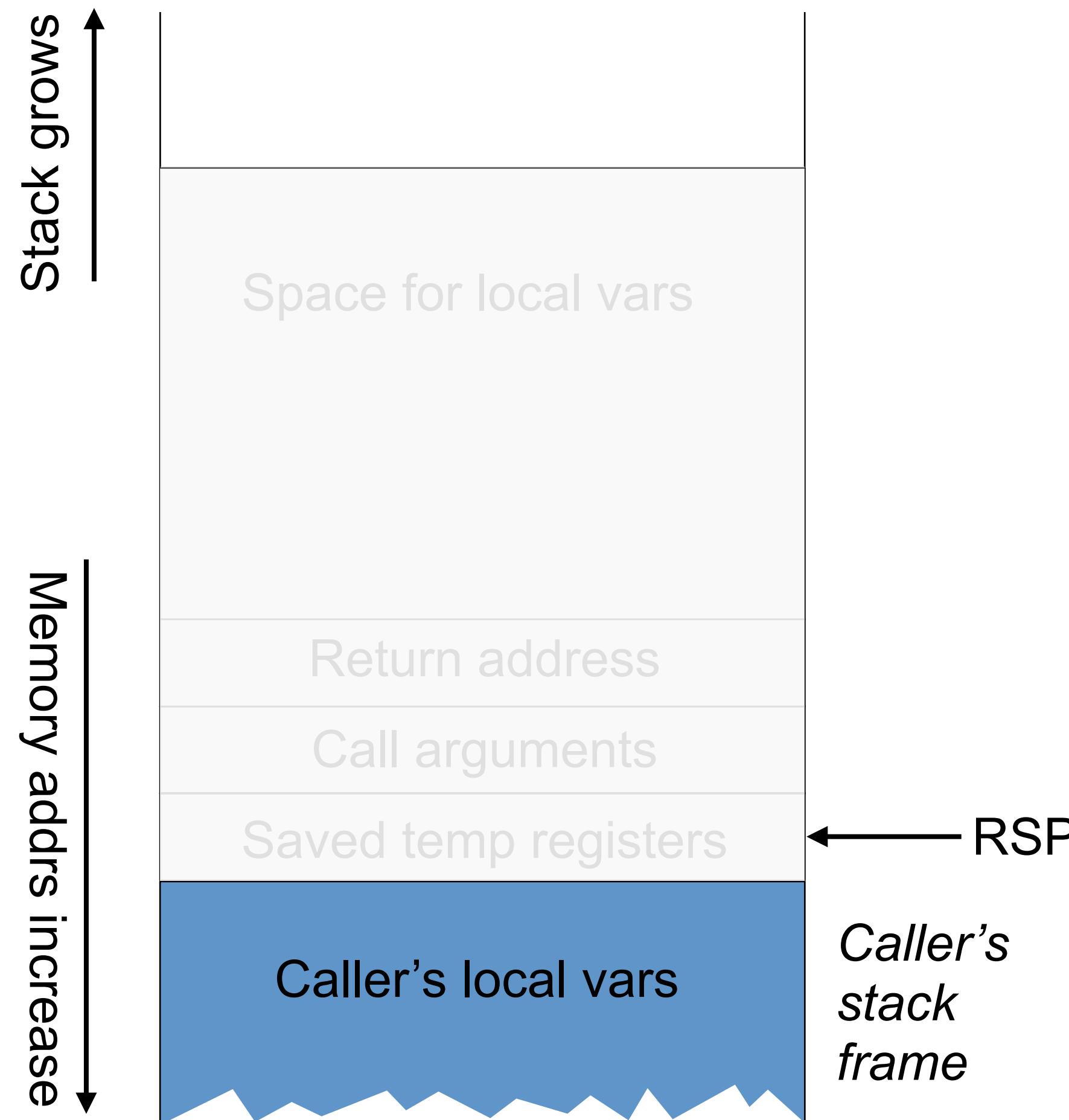


# Stack-based calling convention

- ABI = interface between binary modules

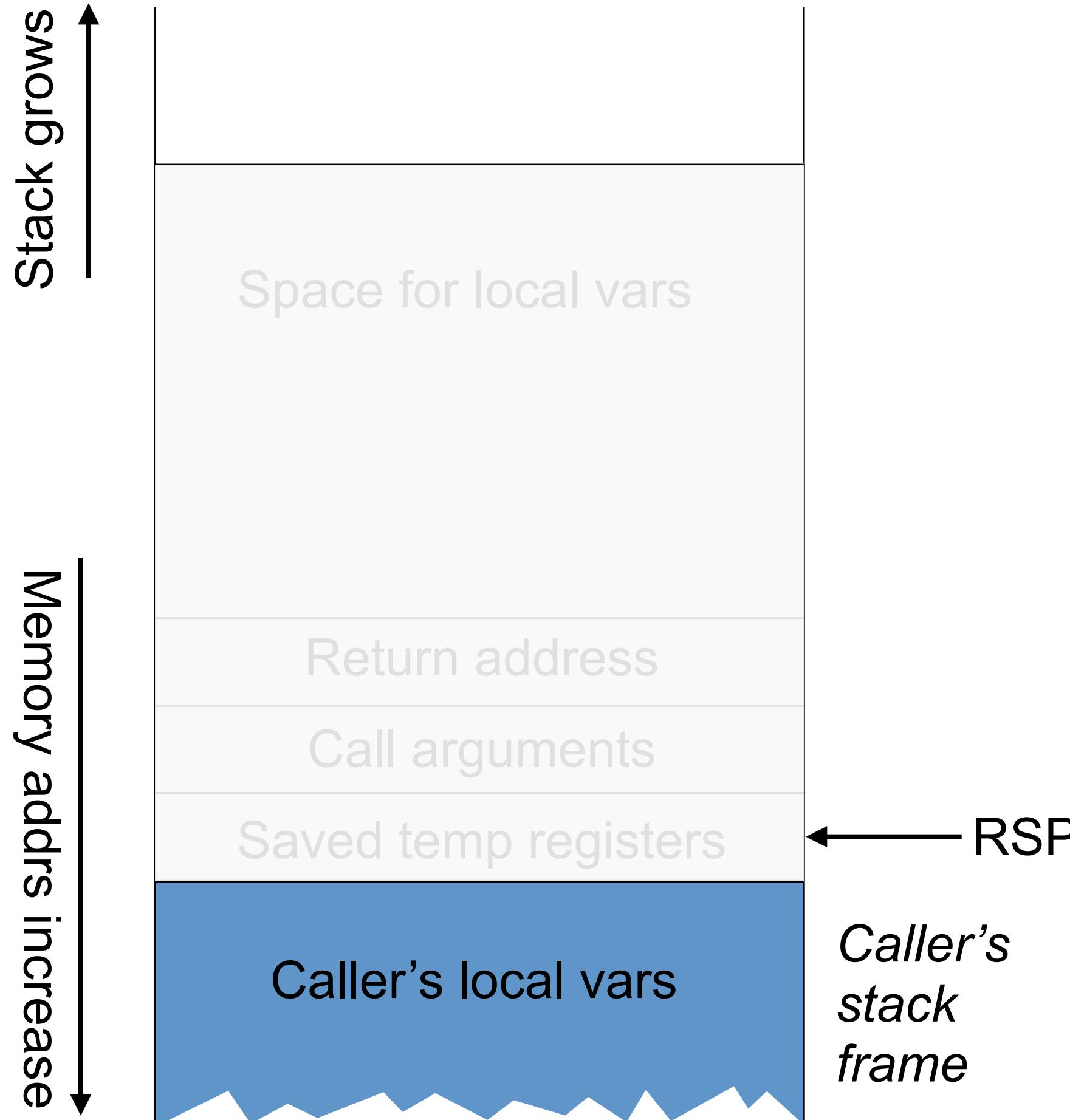


# Stack-based calling convention



- 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

# Stack-based calling convention



- 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

- Local procedure calls (module = procedure) | Memory safety
- Program objects & types (module = memory objects)
- Client/server architecture (different address spaces) | Message-based communication
- Example: Remote procedure calls

# Program Objects & Types

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

# Program objects

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

```
class Rectangle {  
    private int length, width;  
  
    public Rectangle(int l, int b)  
    {  
        length = l;  
        width = b;  
    }  
  
    public int area()  
    {  
        return length * width;  
    }  
}
```

# Program objects

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

*Data separate  
from Behavior*

vs.

*Data + Behavior  
inseparable*

```
class Rectangle {  
  
    private int length, width;  
  
    public Rectangle(int l, int b)  
    {  
        length = l;  
        width = b;  
    }  
  
    public int area()  
    {  
        return length * width;  
    }  
}
```

# Program objects

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

*Data separate  
from Behavior*

*vs.*

*Data + Behavior  
inseparable*

*Encapsulation*

```
class Rectangle {  
    private int length, width;  
  
    public Rectangle(int l, int b)  
    {  
        length = l;  
        width = b;  
    }  
  
    public int area()  
    {  
        return length * width;  
    }  
}
```

# Objects & type safety = stronger intra-program modularity

- Untyped languages



# 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)*



# 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*



# 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*

# 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...)*

# 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...)*

# 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

- Recap of modularization
  - Local procedure calls (module = procedure)
  - Program objects & types (module = memory object)
  - Client/server architecture (different address spaces)
  - Example: Remote procedure calls
- 
- Memory safety
- Message-based communication

# Clients/Servers Interacting via Messages

*Modularization across different address spaces*

# Splitting into Clients and Servers

---

# Splitting into Clients and Servers

---

---

- Is the foundation for many system architecture patterns
  - *event-driven, microservices/SOA, action–domain–responder (e.g., MVVM), multi-tiered, peer-to-peer, publish-subscribe, etc.*

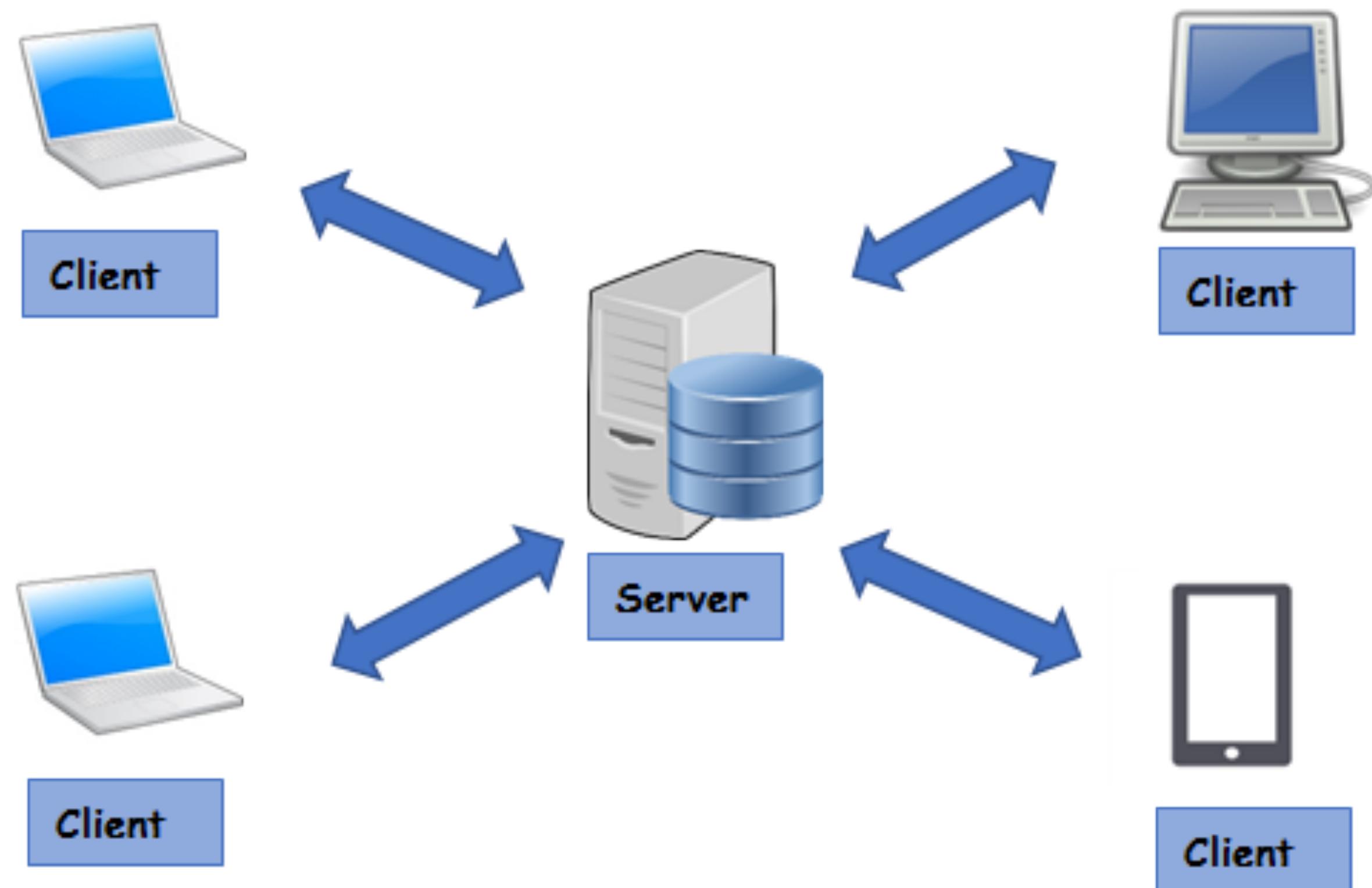
# Splitting into Clients and Servers

---

- Is the foundation for many system architecture patterns
  - *event-driven, microservices/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

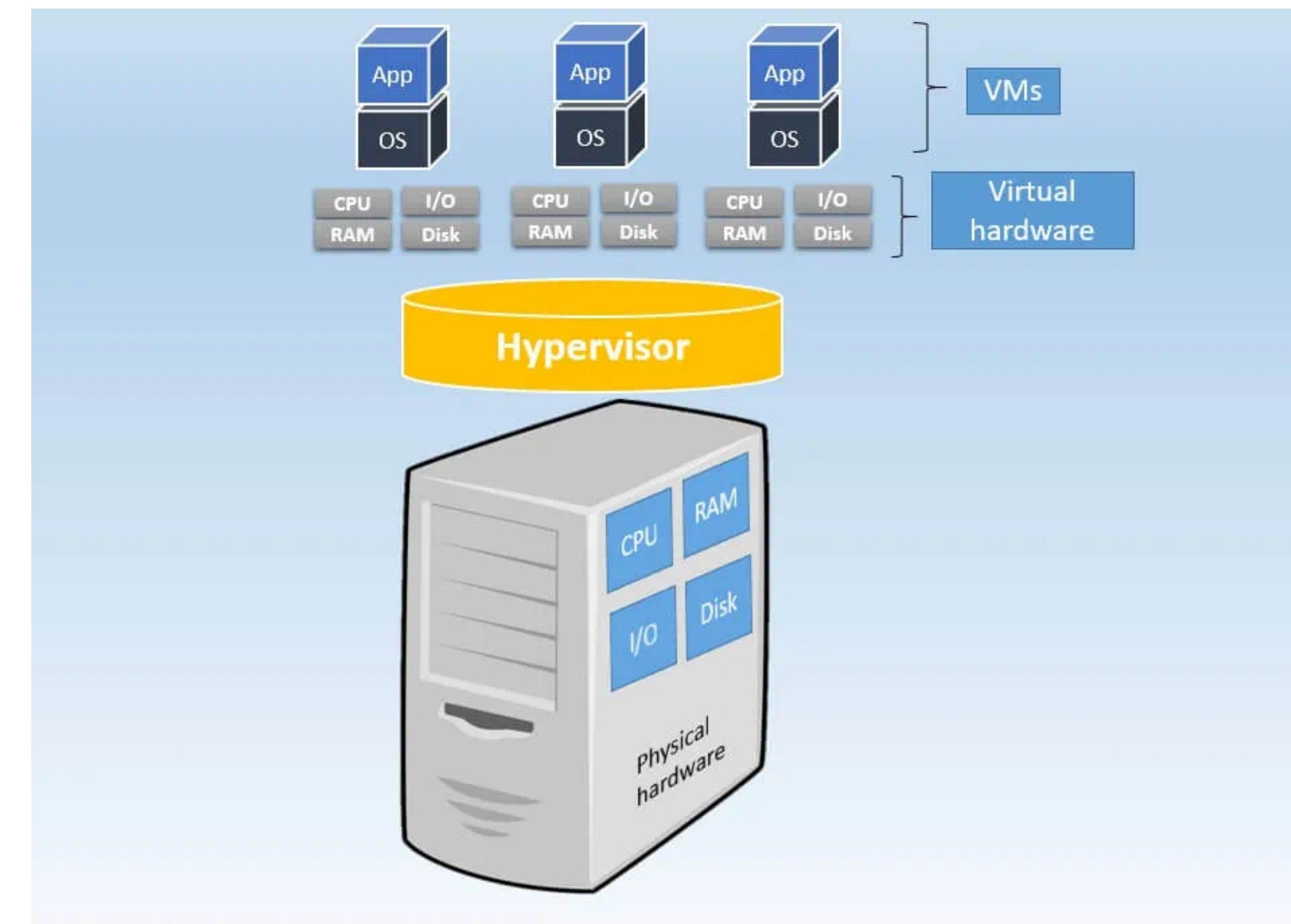
- Rely on physics
- Reduce fate sharing
- Improve encapsulation



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

# Physical (and virtual) servers

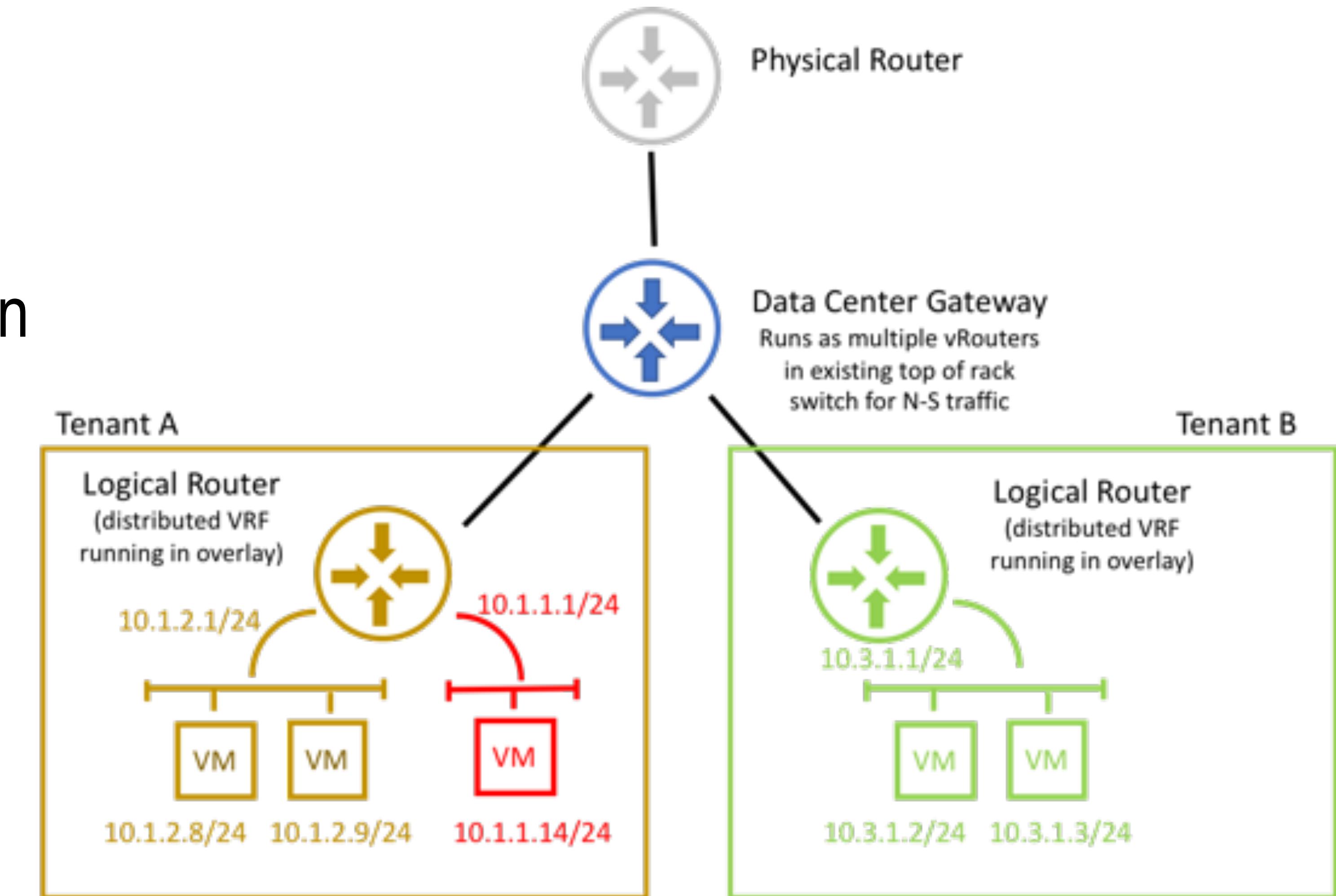
- Rely on physics
- Reduce fate sharing
- Improve encapsulation



<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

---

# Microkernels

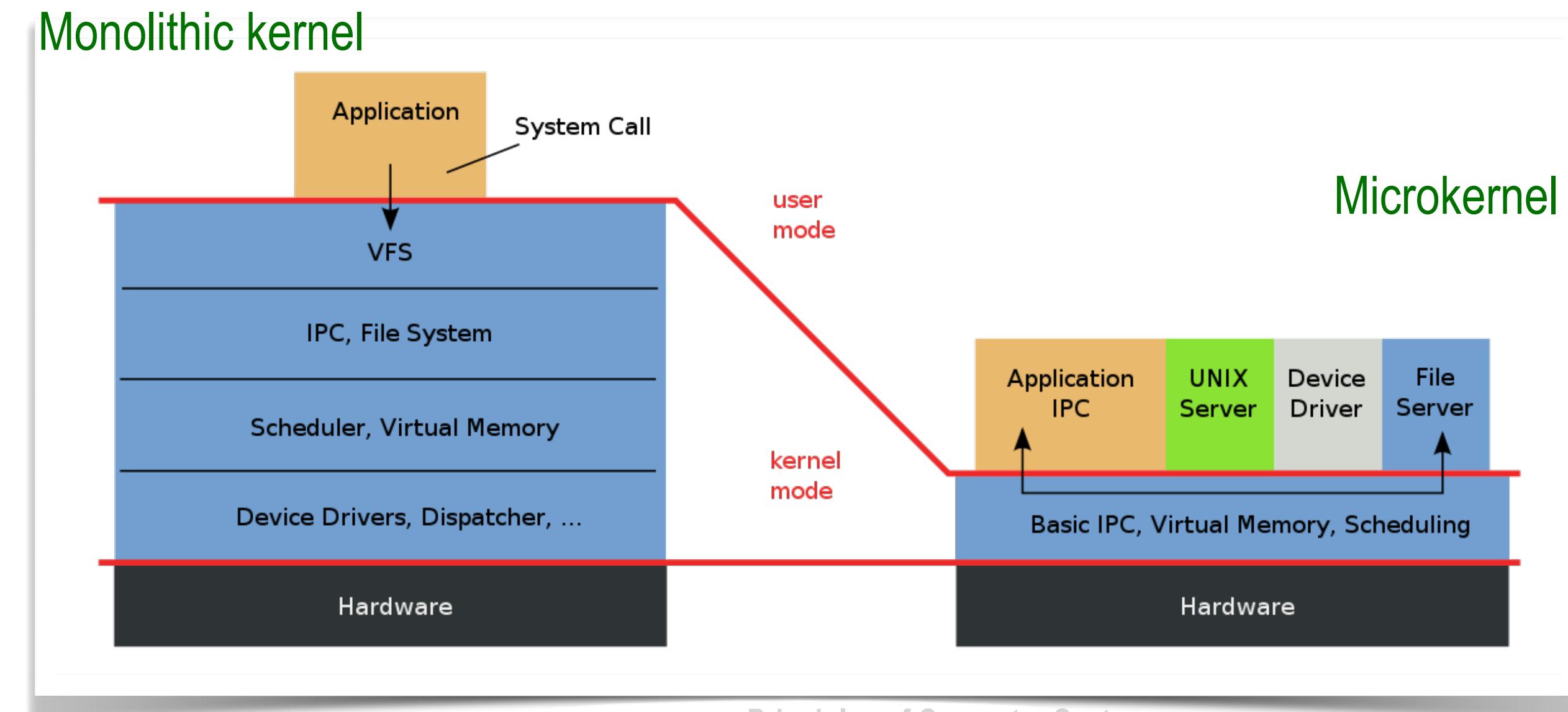
---

---

- An exercise in modularization of otherwise monolithic kernels
  - *Liedtke's minimality principle*

# 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*

# 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”*

# 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*

# 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

# Drawbacks of Client/Server

---

---

- Marshalling/unmarshalling messages incurs overheads
- Unnatural interaction between modules

# Drawbacks of Client/Server

---

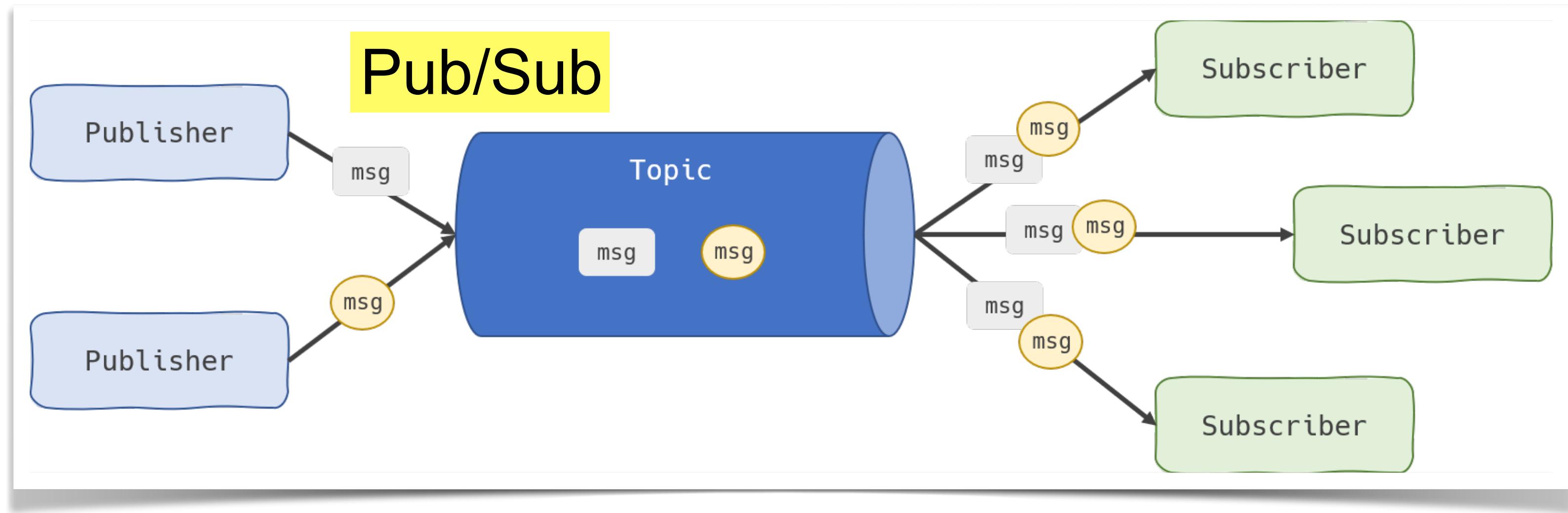
- Marshalling/unmarshalling messages incurs overheads
- Unnatural interaction between modules
- Semantic coupling may render functional decoupling moot
  - *E.g., caller cannot make progress without an answer*

# 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](https://microservices.io/i/Microservice_Architecture.png)

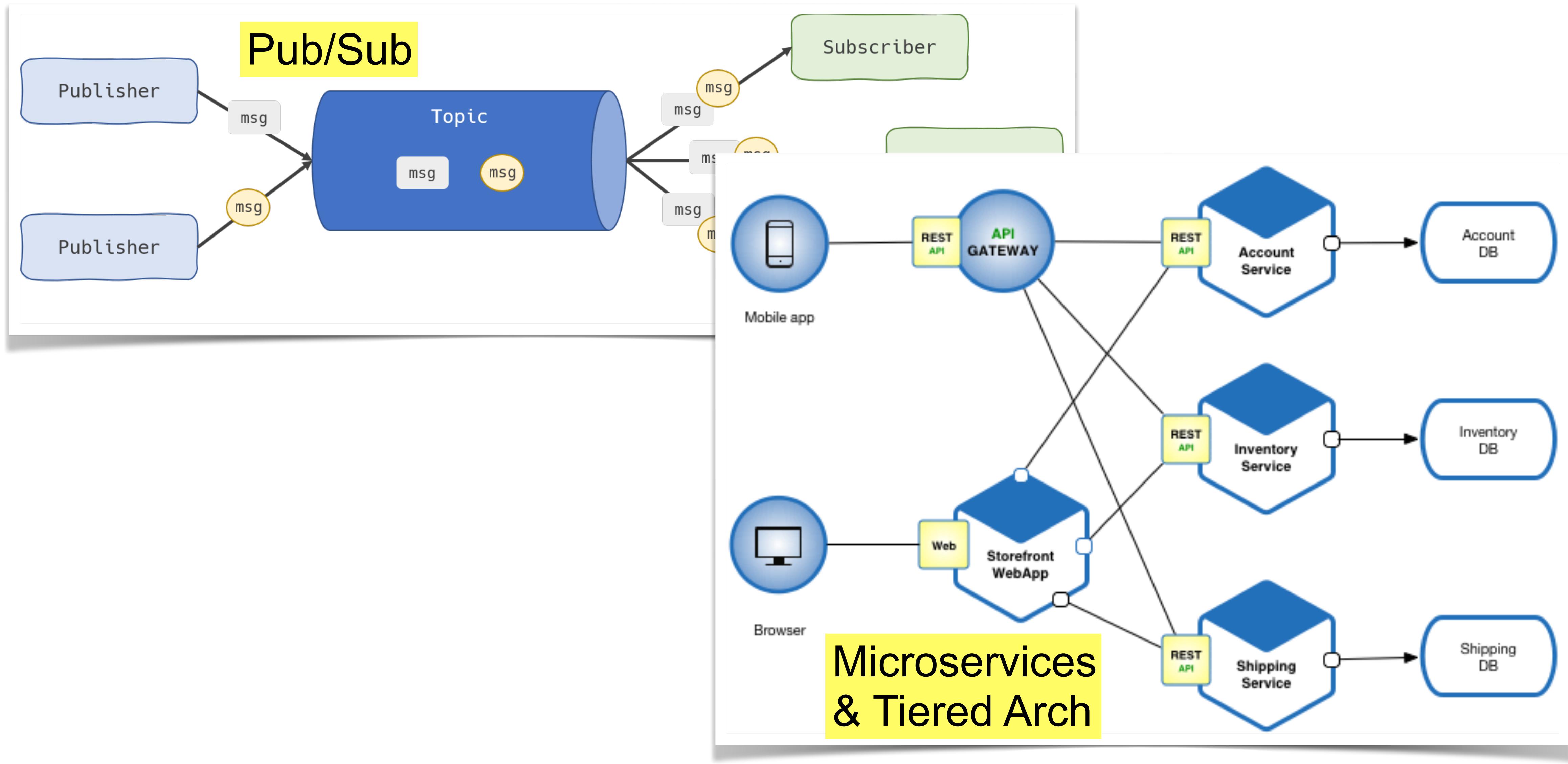
# A couple of examples of client/server architectures



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

[https://microservices.io/i/Microservice\\_Architecture.png](https://microservices.io/i/Microservice_Architecture.png)

# A couple of examples of client/server architectures



# Outline

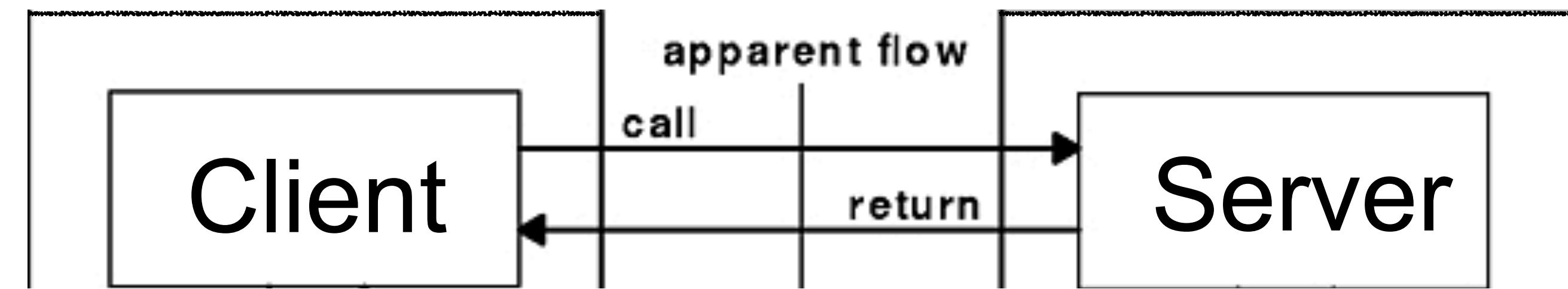
- Local procedure calls (module = procedure) | Memory safety
- Program objects & types (module = memory object)
- Client/server architecture (different address spaces) | Message-based communication
- Example: Remote procedure calls



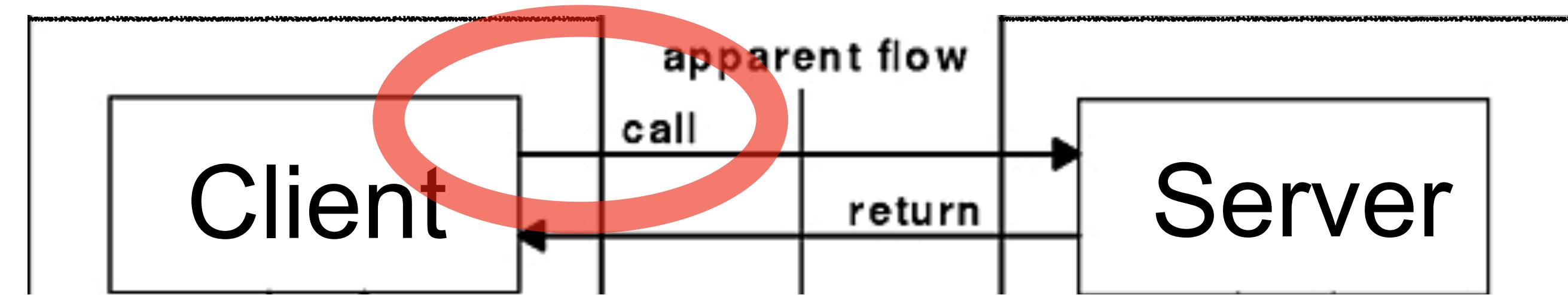
# Remote Procedure Calls (RPC)

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

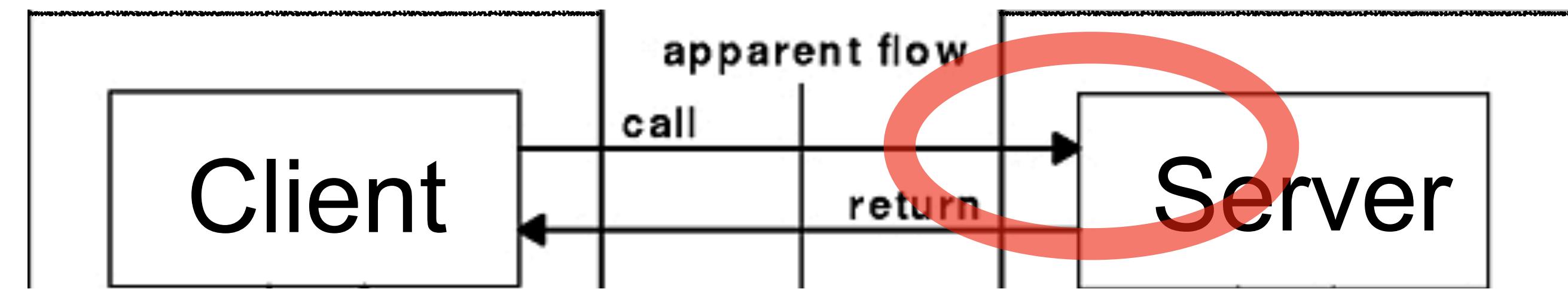
# Mechanics of RPC



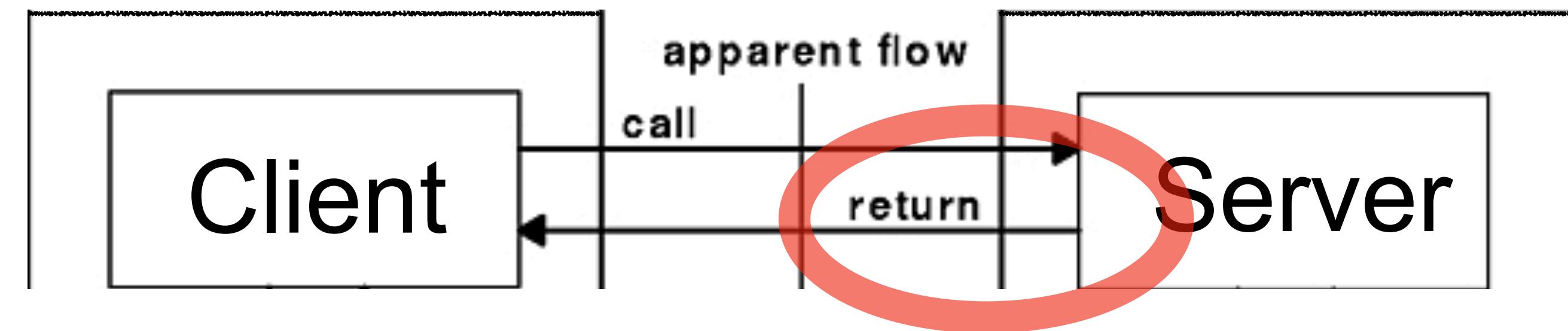
# Mechanics of RPC



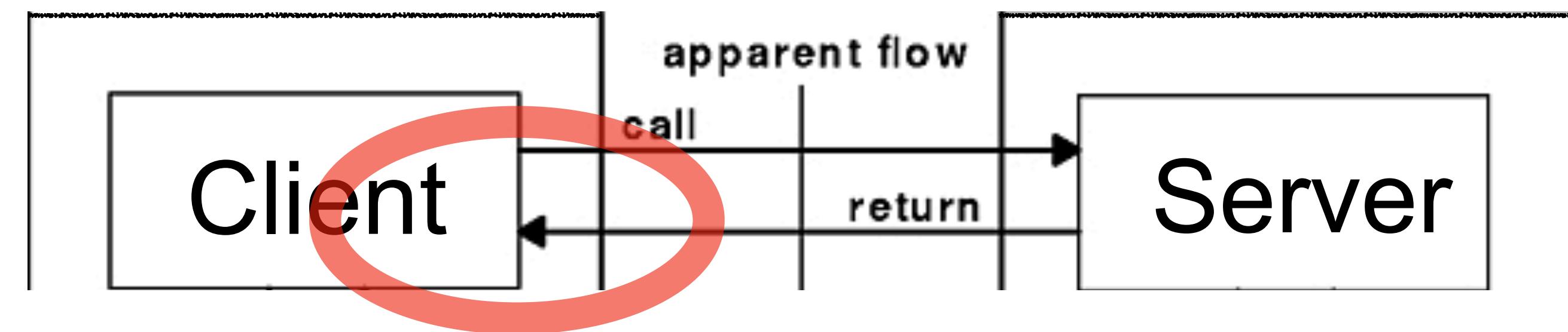
# Mechanics of RPC



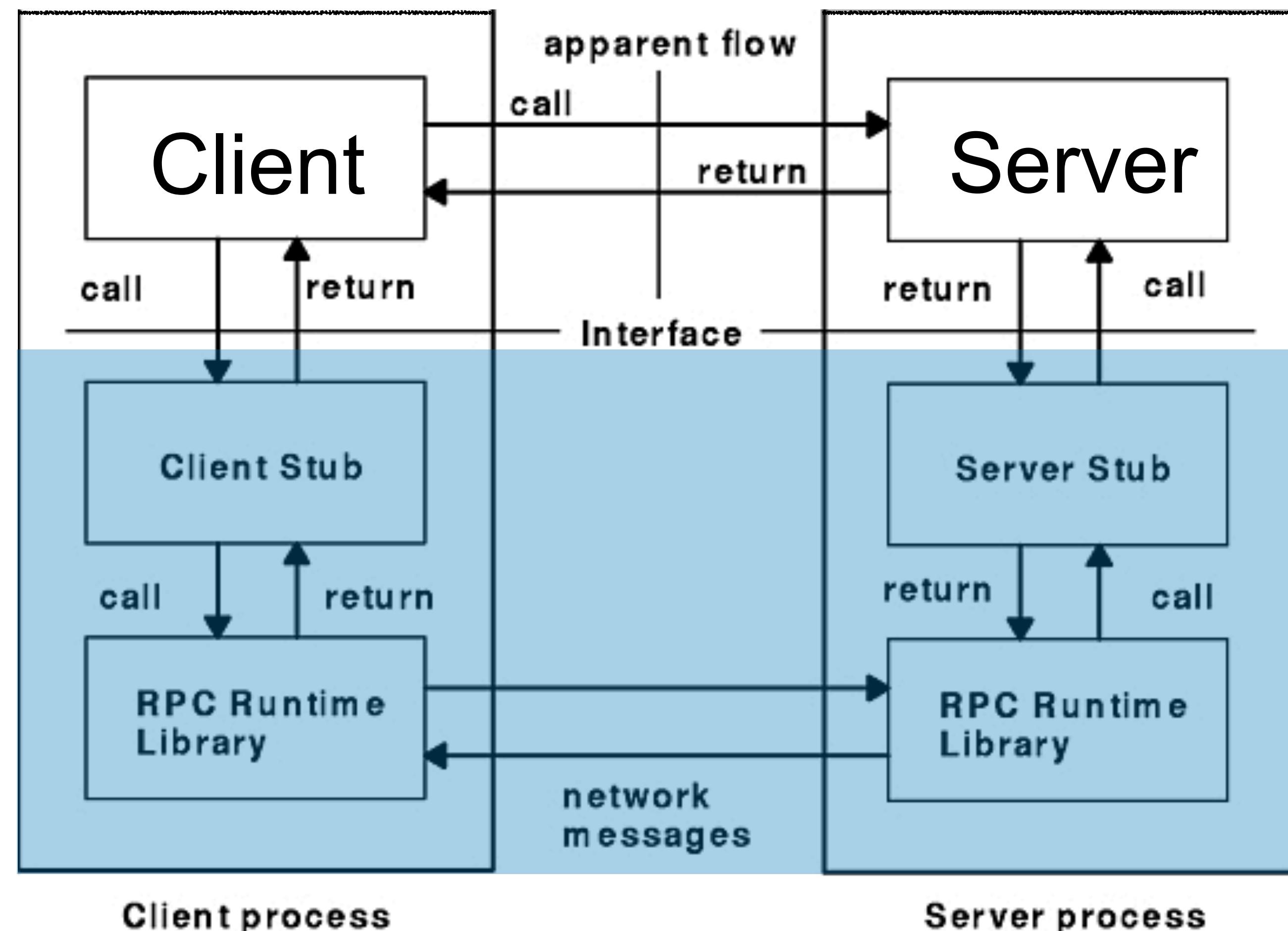
# Mechanics of RPC



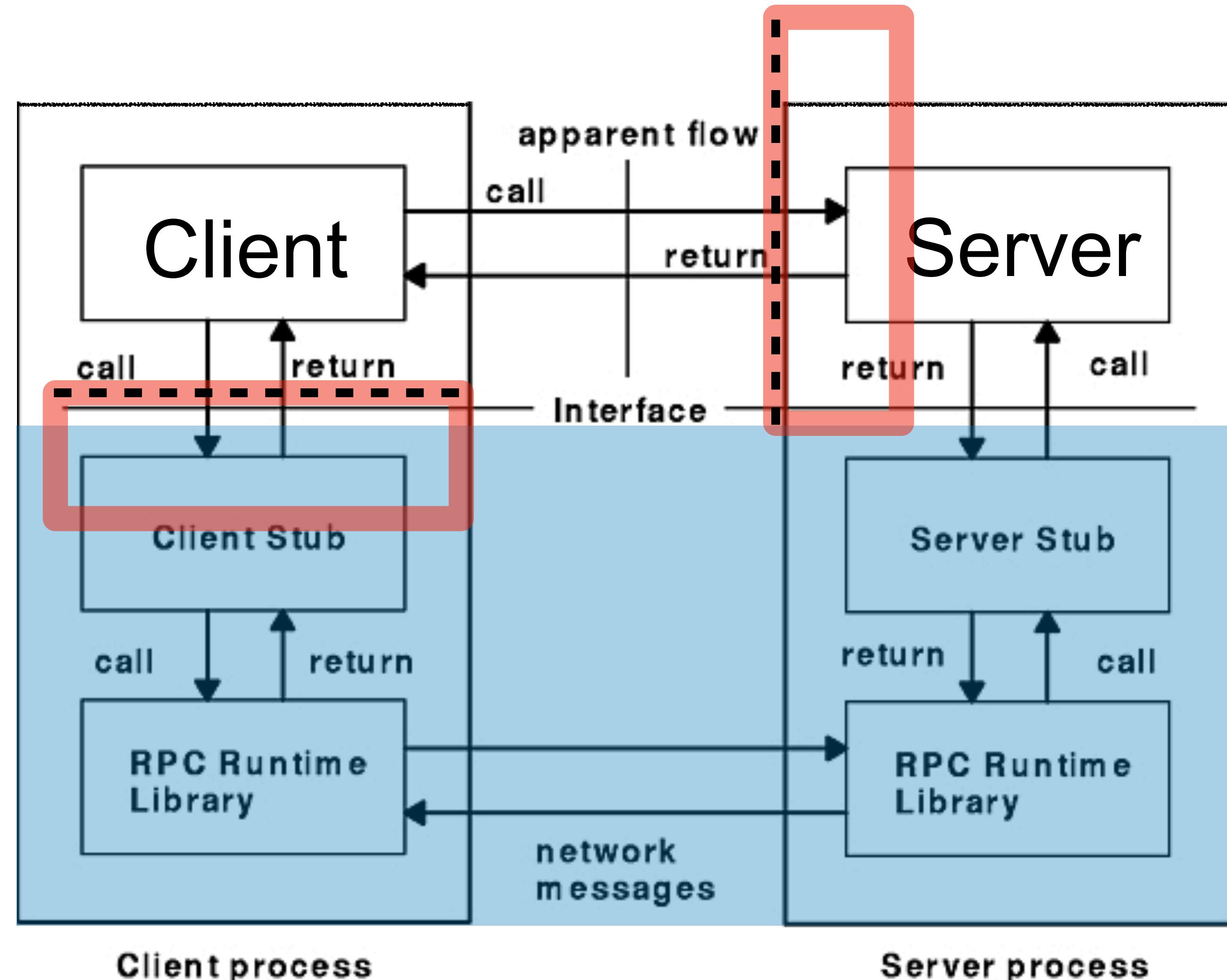
# Mechanics of RPC



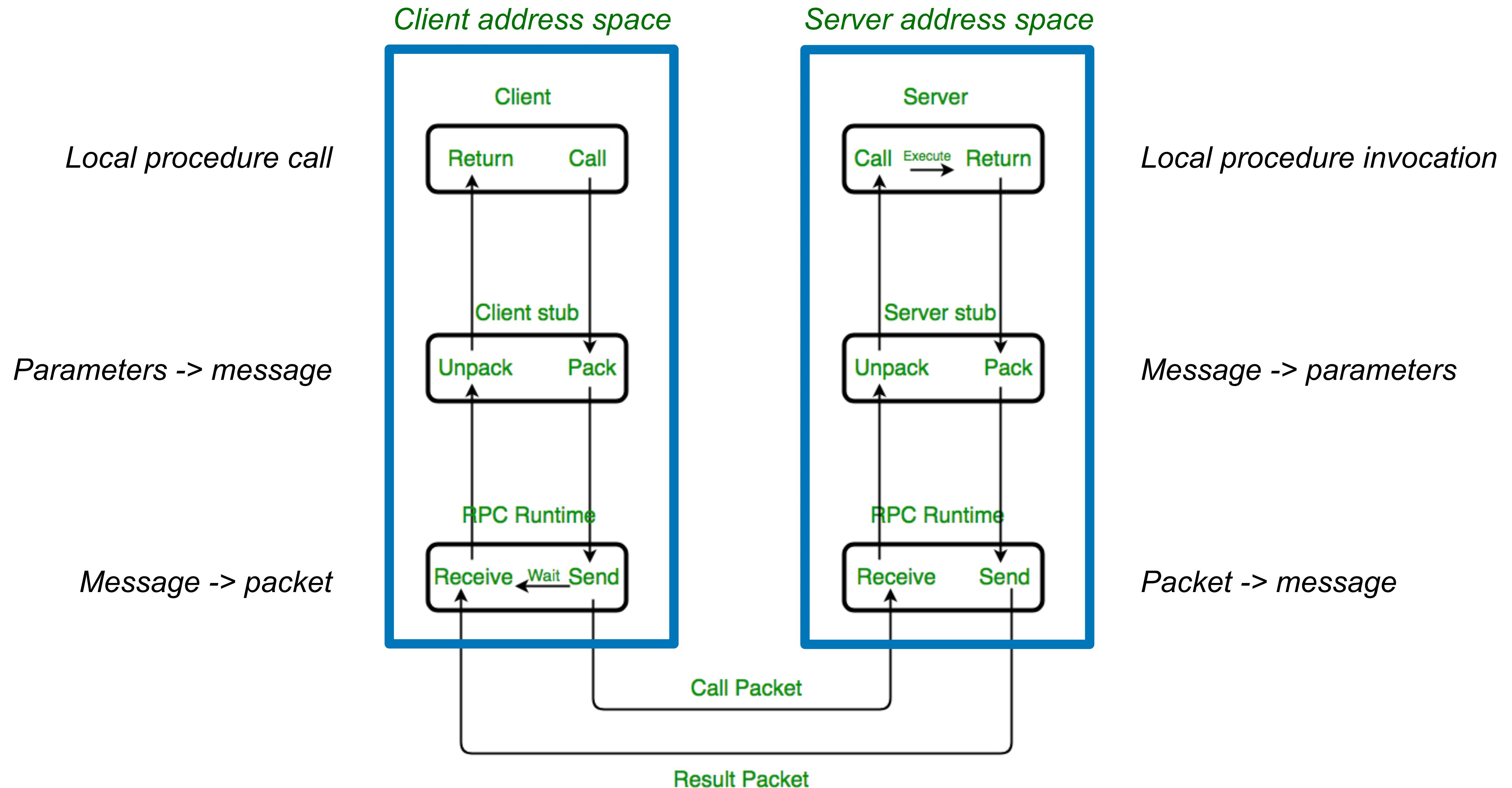
# Mechanics of RPC



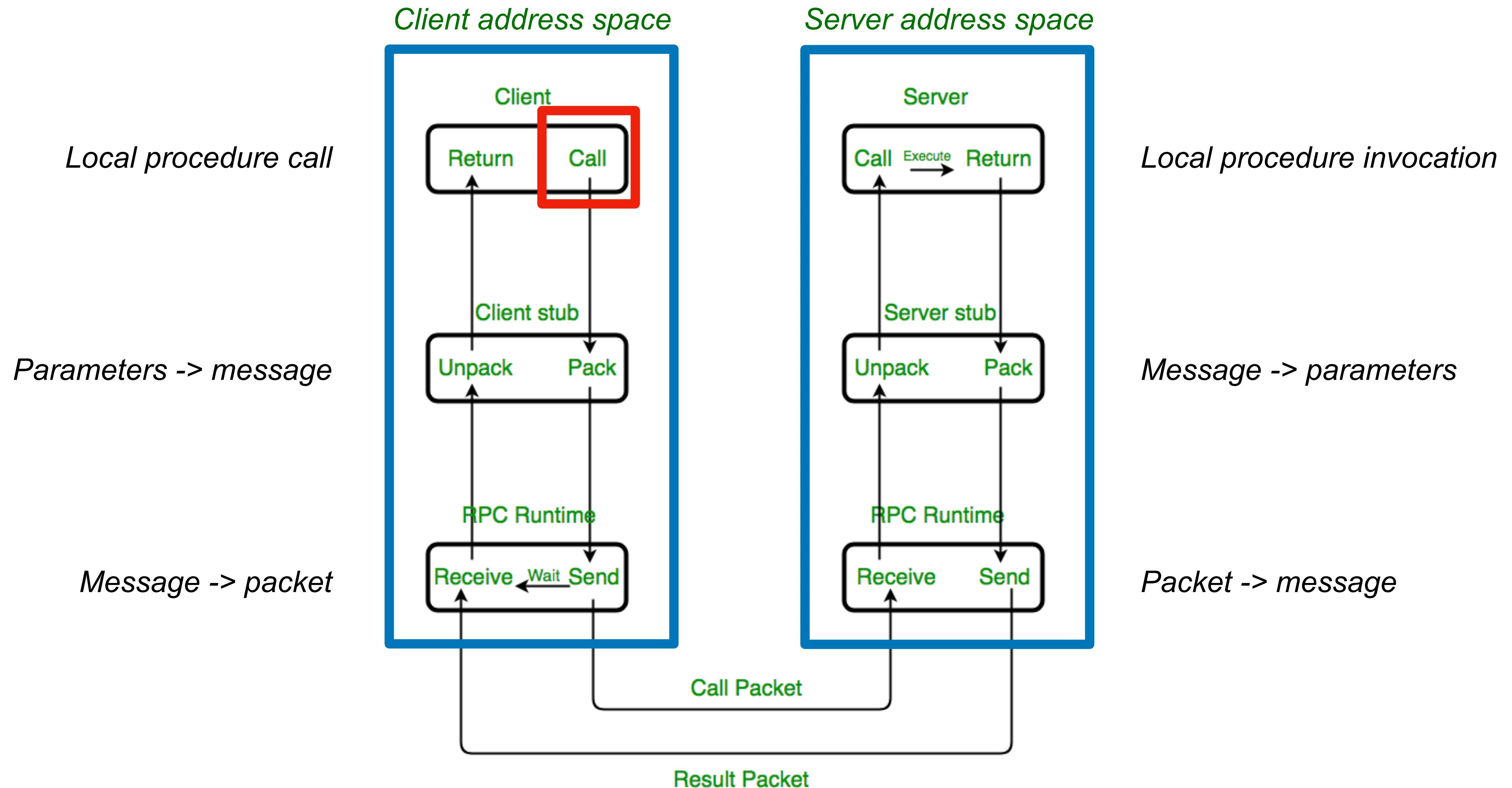
# Mechanics of RPC



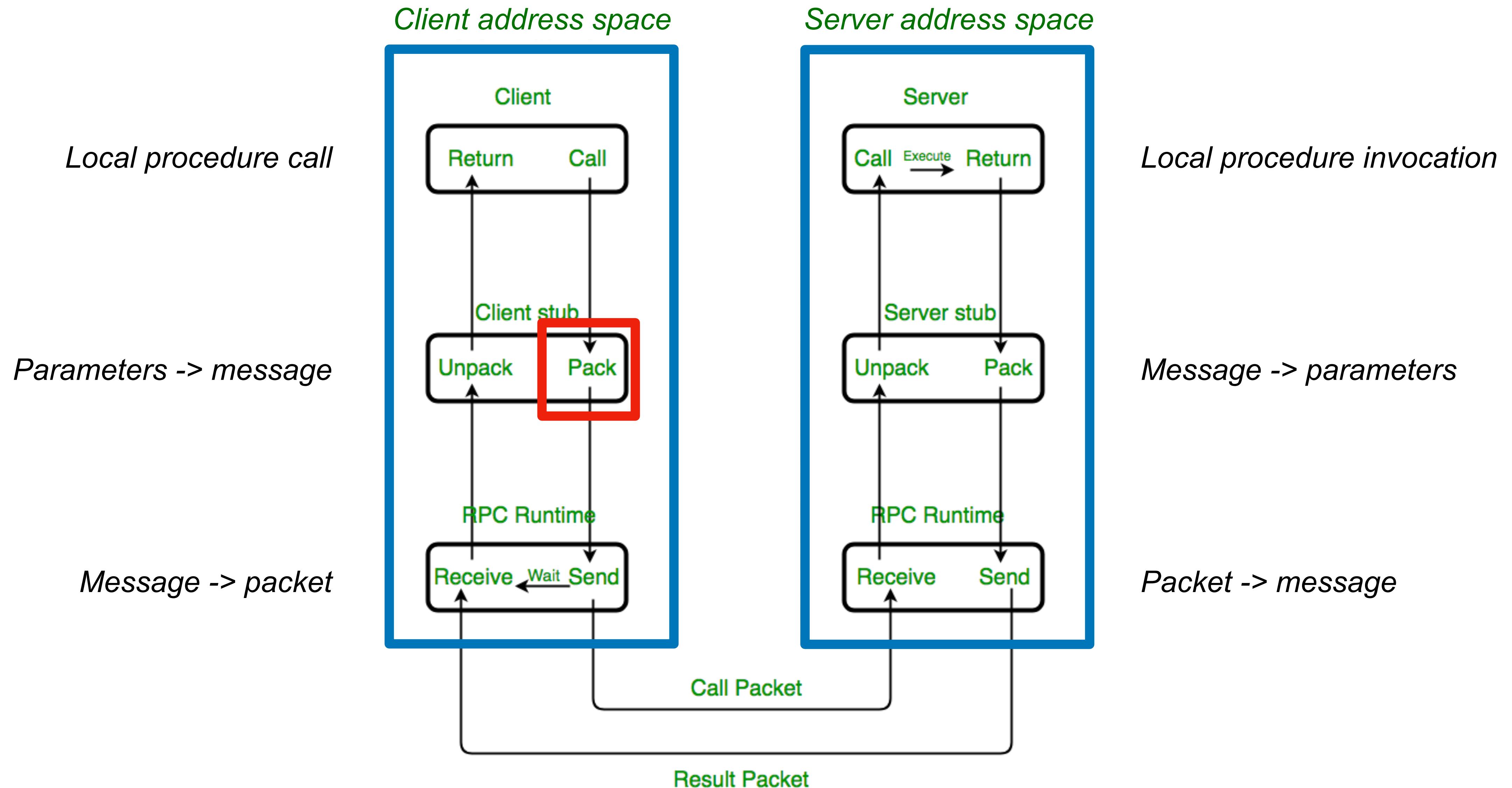
# Mechanics of RPC



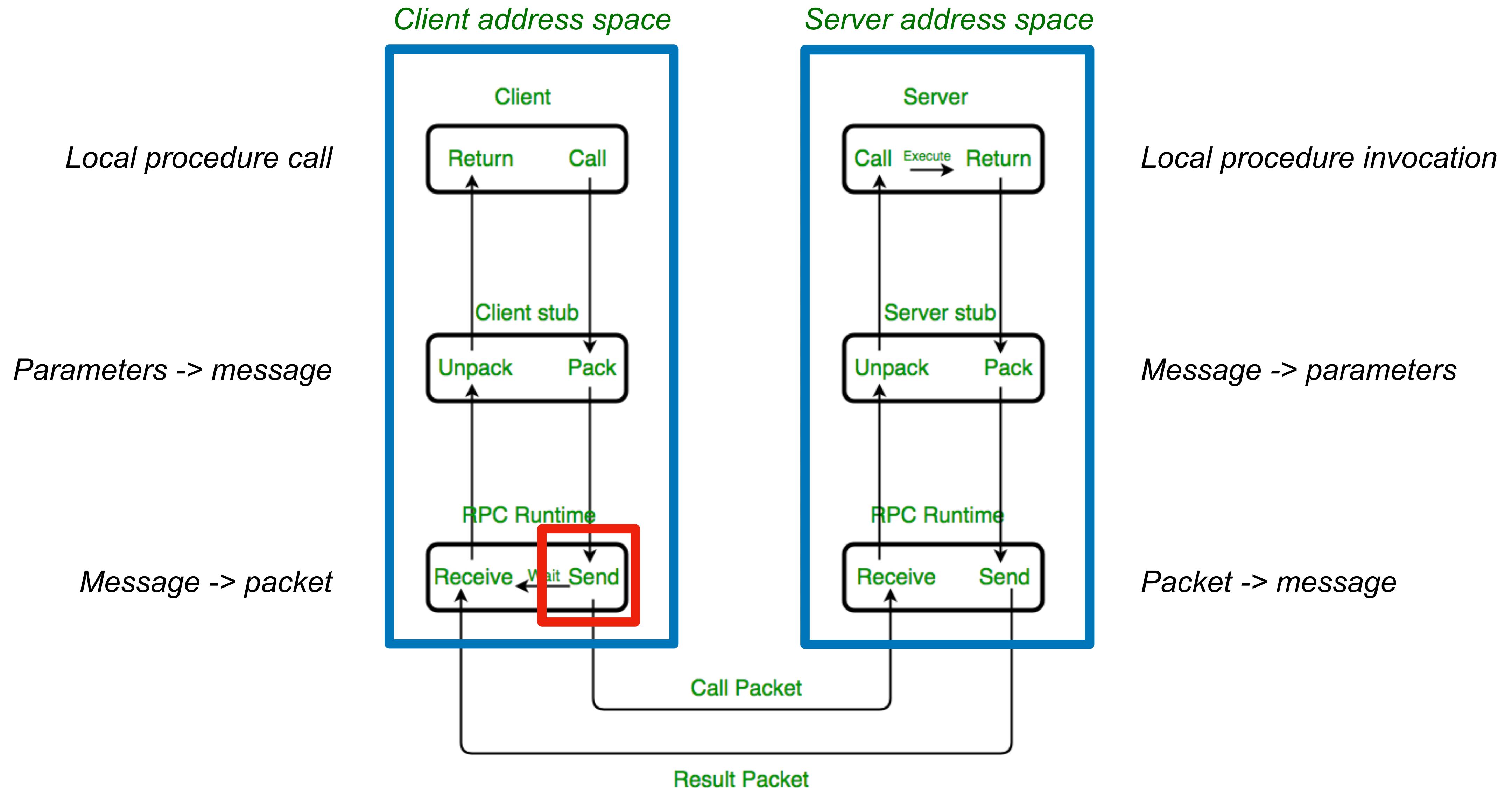
# Mechanics of RPC



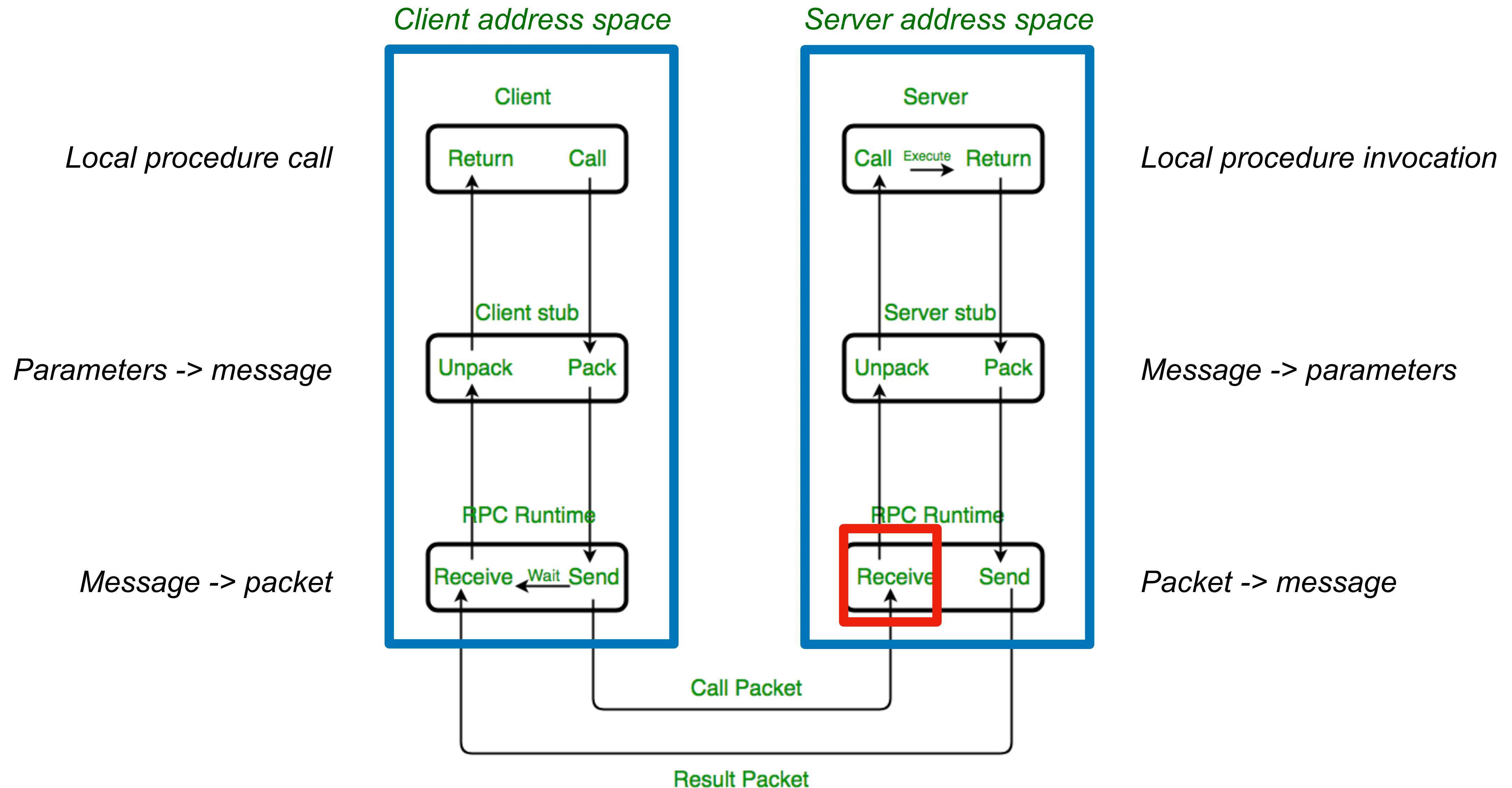
# Mechanics of RPC



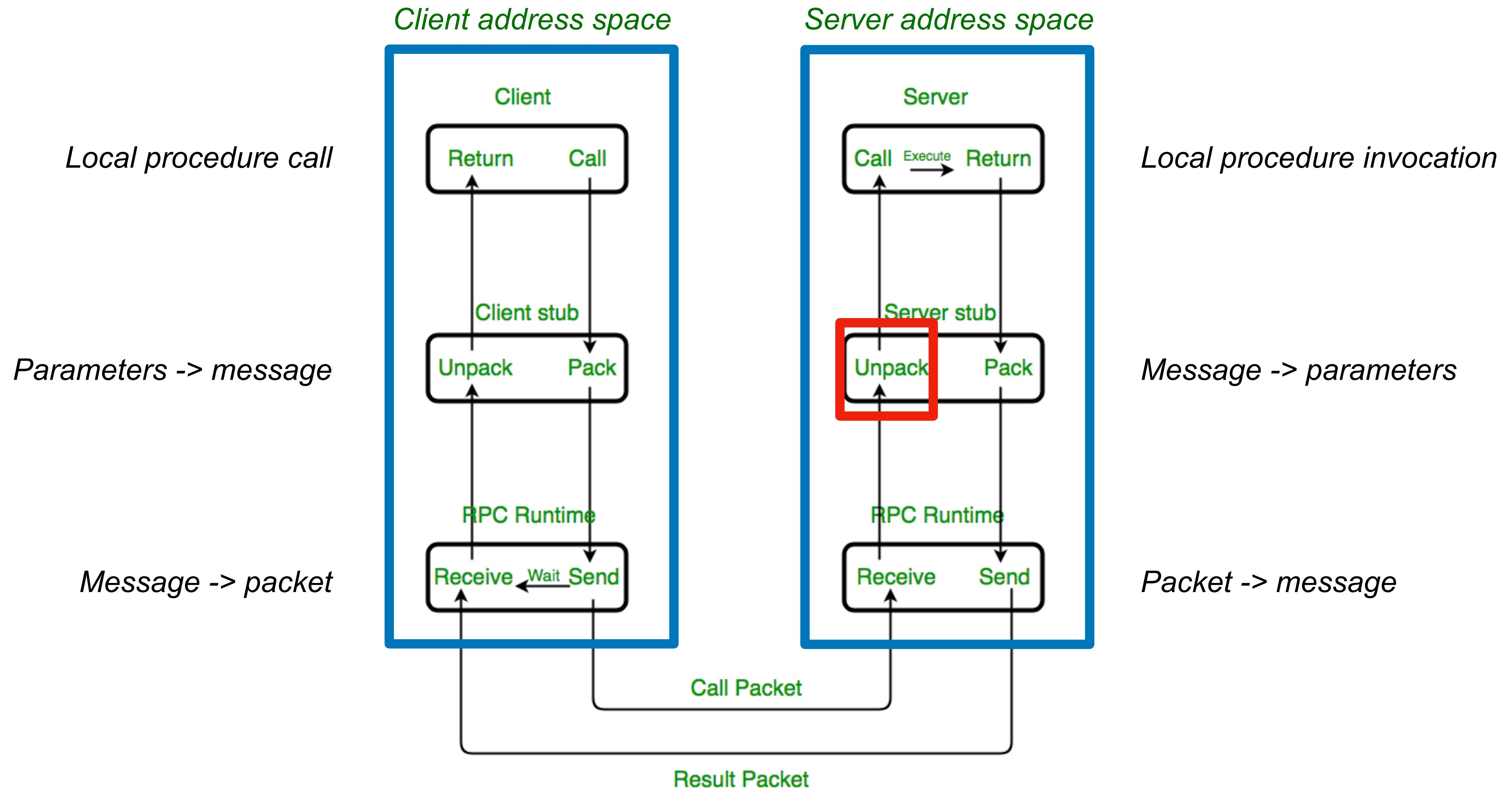
# Mechanics of RPC



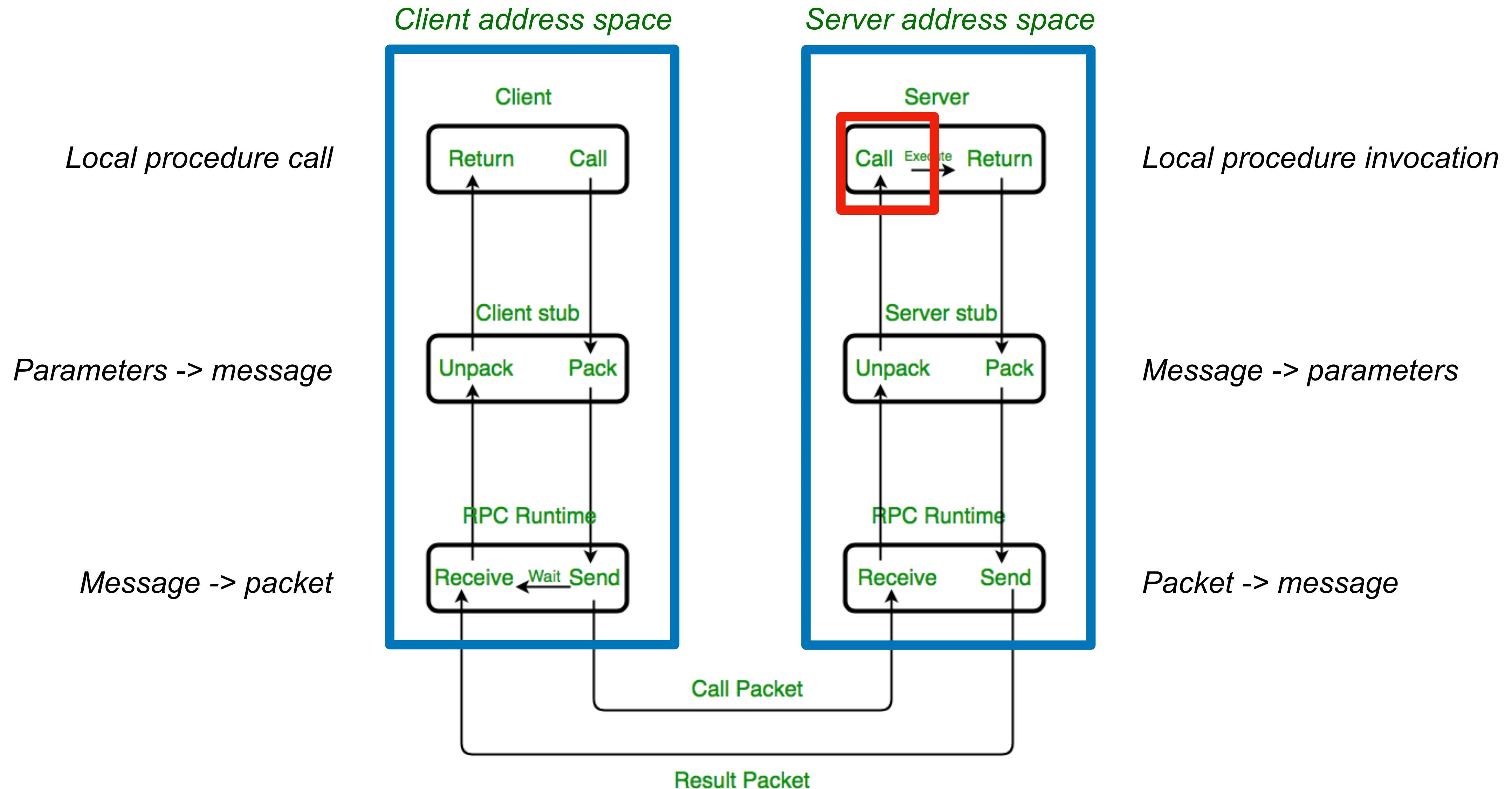
# Mechanics of RPC



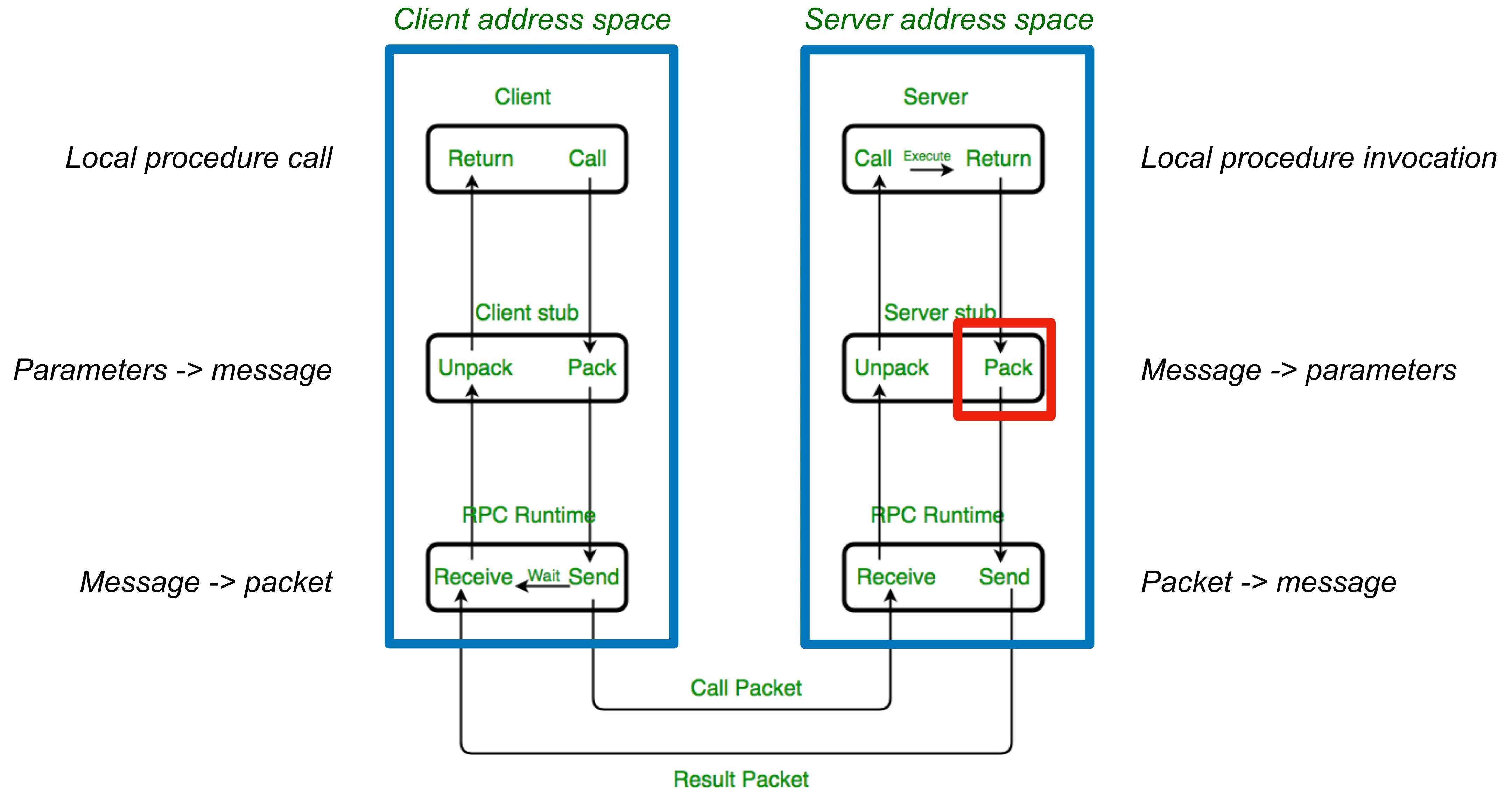
# Mechanics of RPC



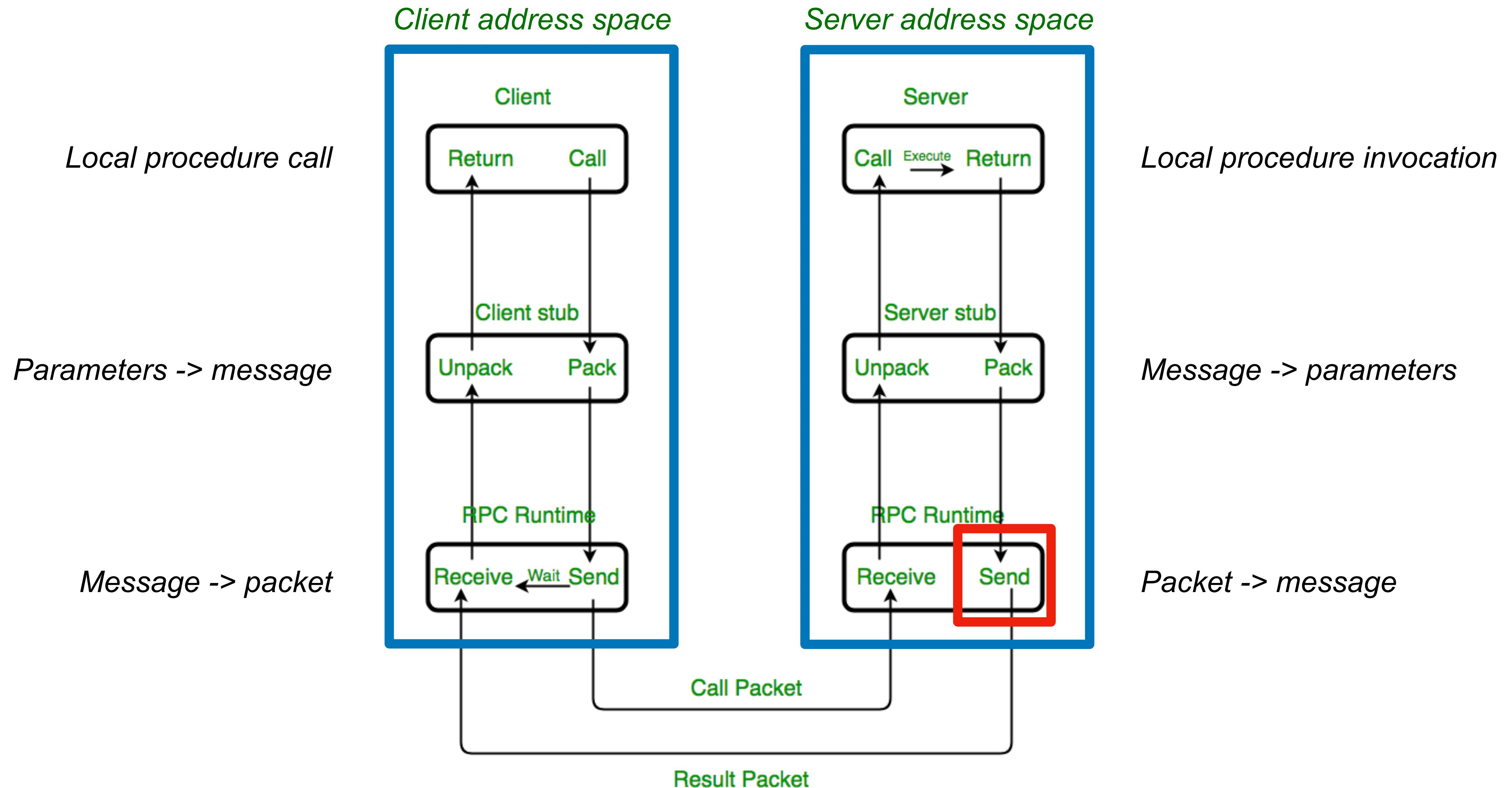
# Mechanics of RPC



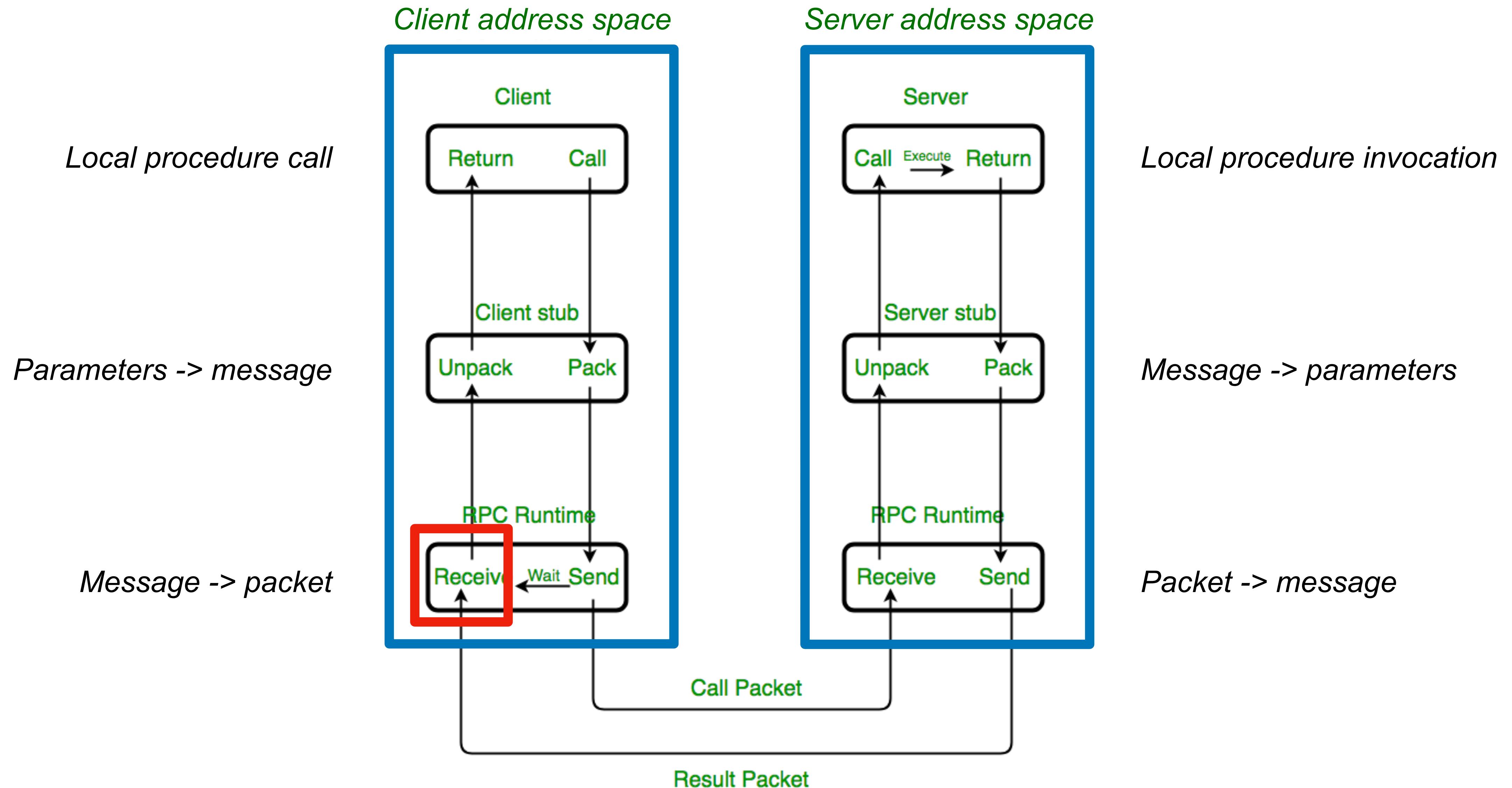
# Mechanics of RPC



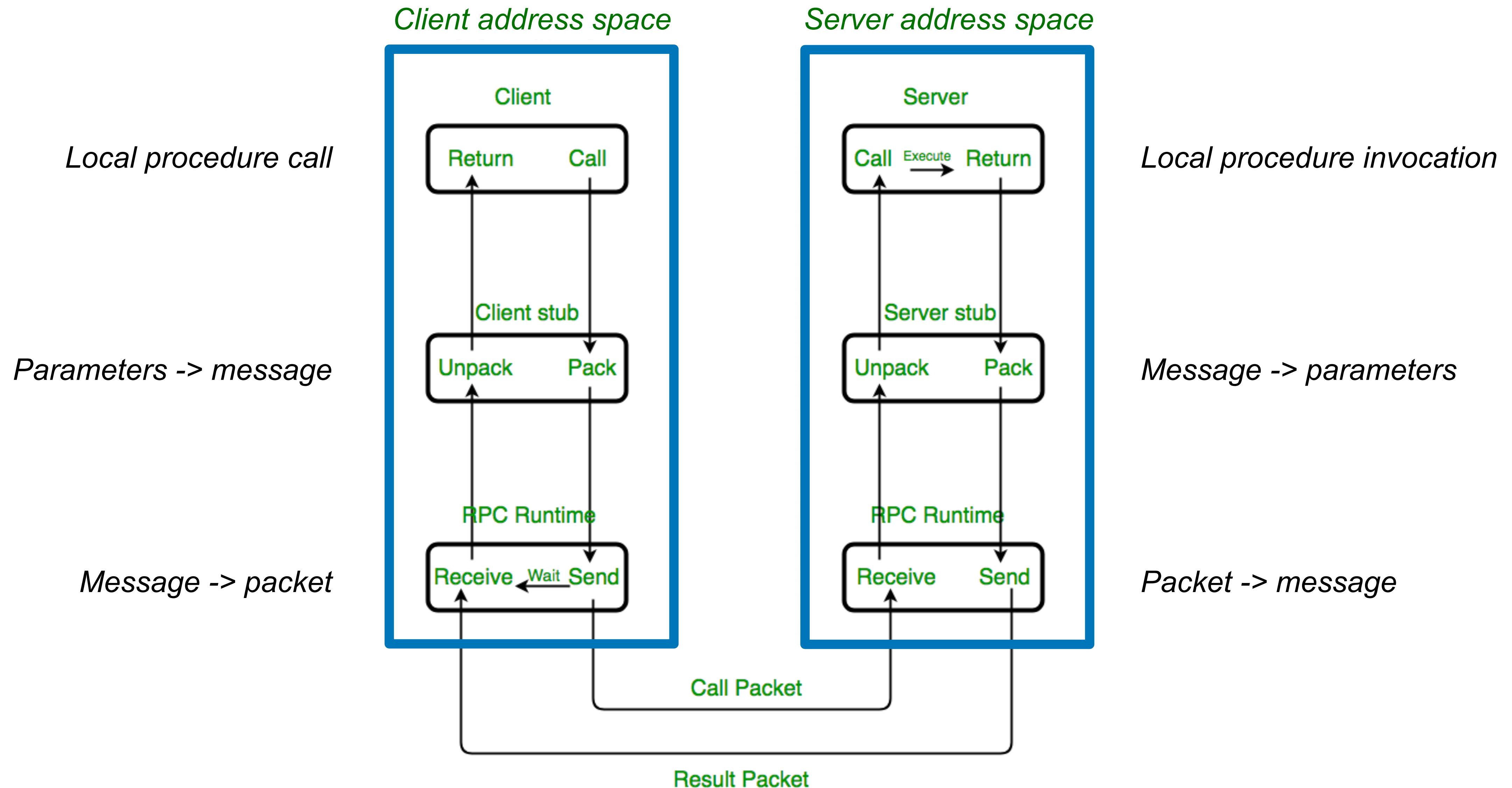
# Mechanics of RPC



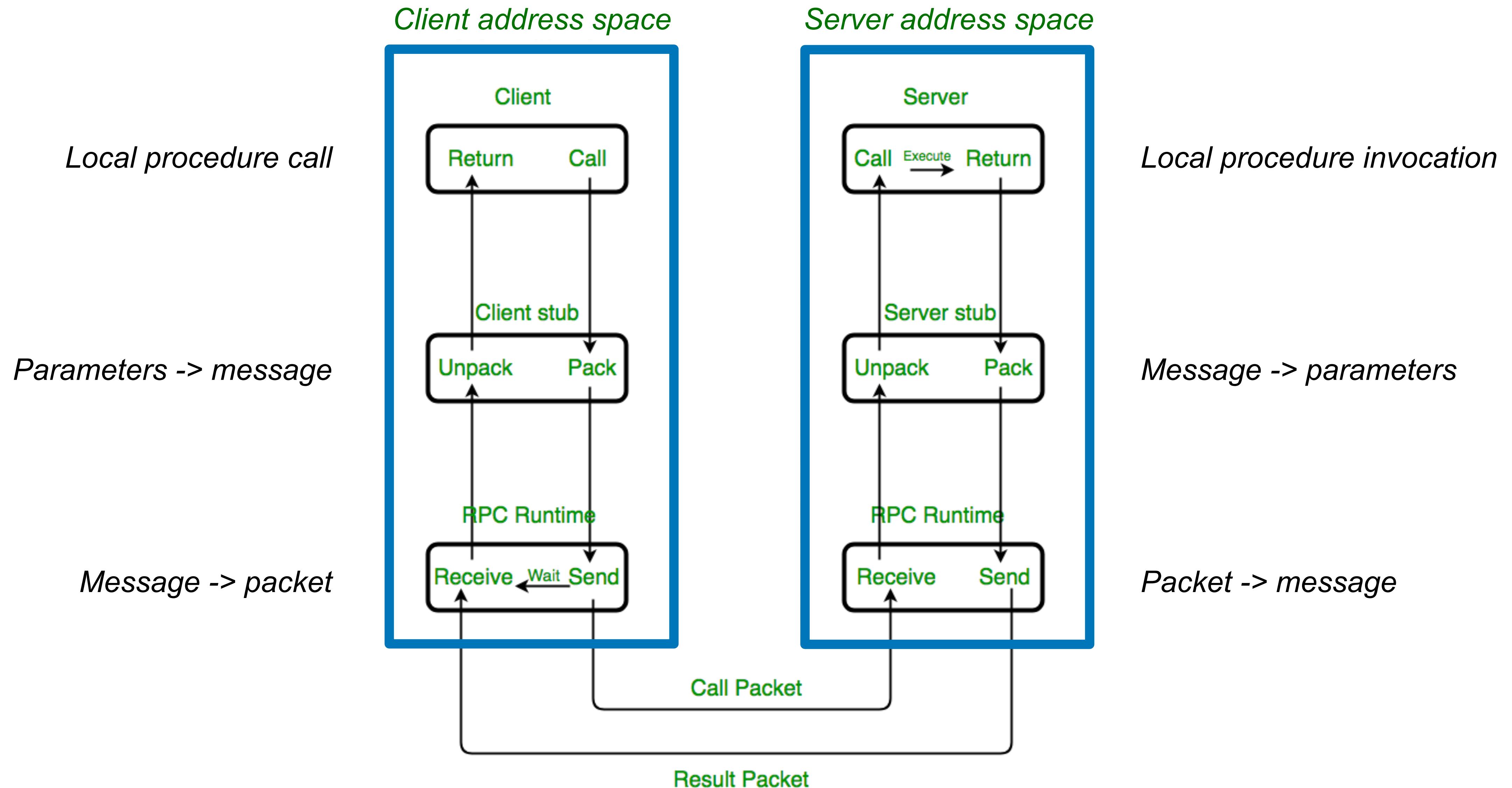
# Mechanics of RPC



# Mechanics of RPC



# Mechanics of RPC



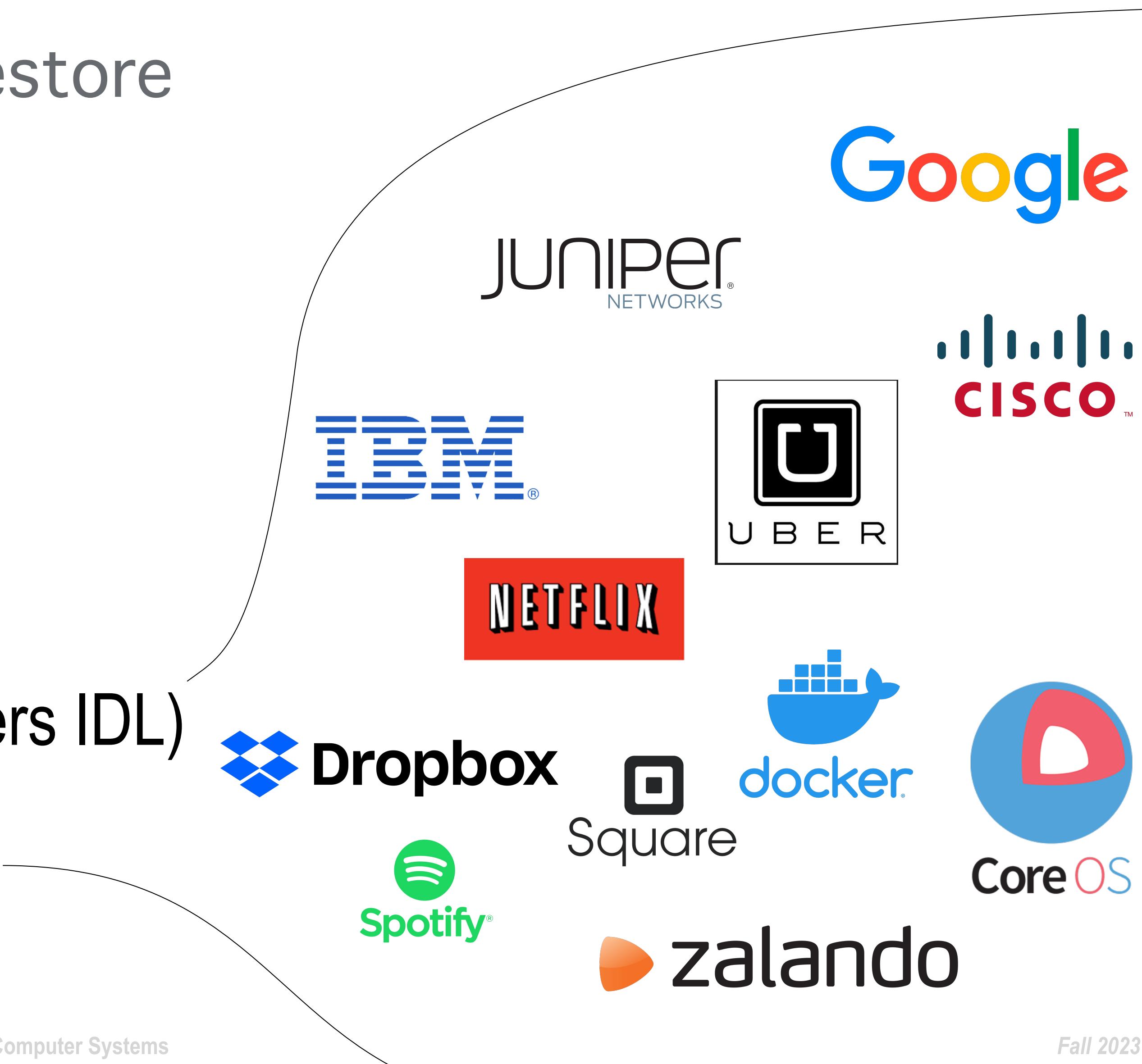
# Examples of RPC systems

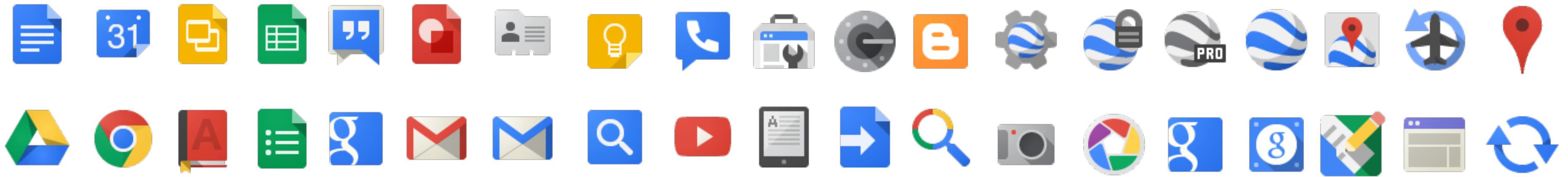
---

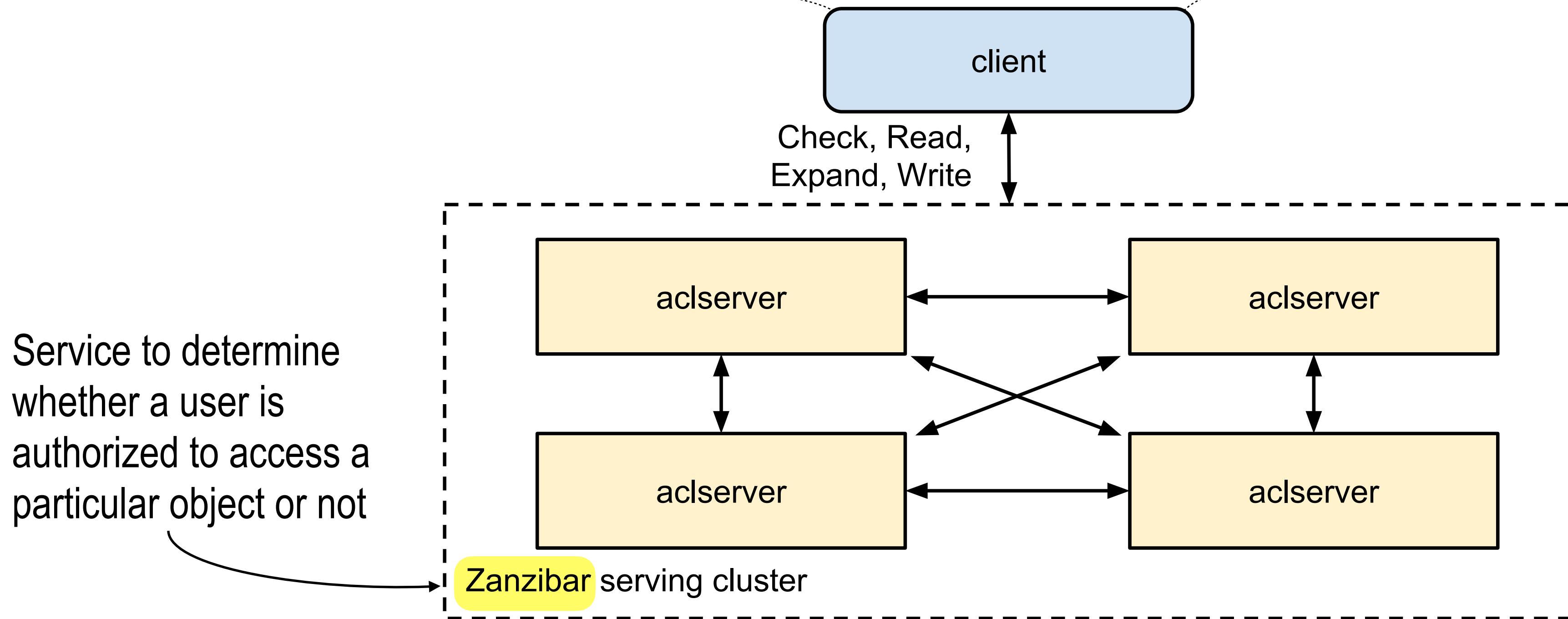
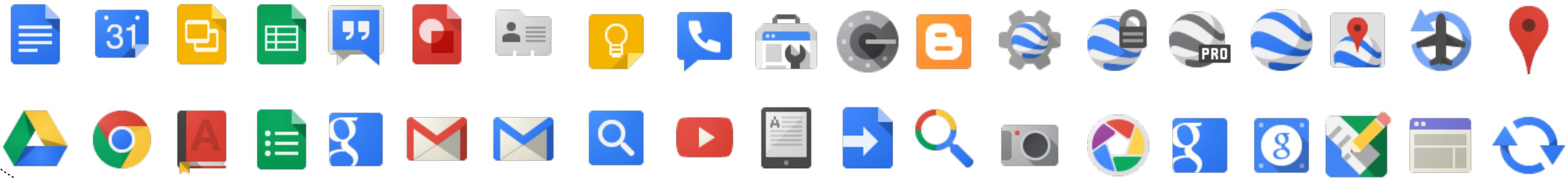
- NFS and  Google Cloud Filestore
- Java RMI and Google Web Toolkit
- Go's rpc package
- Cassandra, HBase, Couchbase
- Apache Thrift
- gRPC (uses Google Protocol Buffers IDL)
  - *microservices, mobile, real-time, IoT, ...*

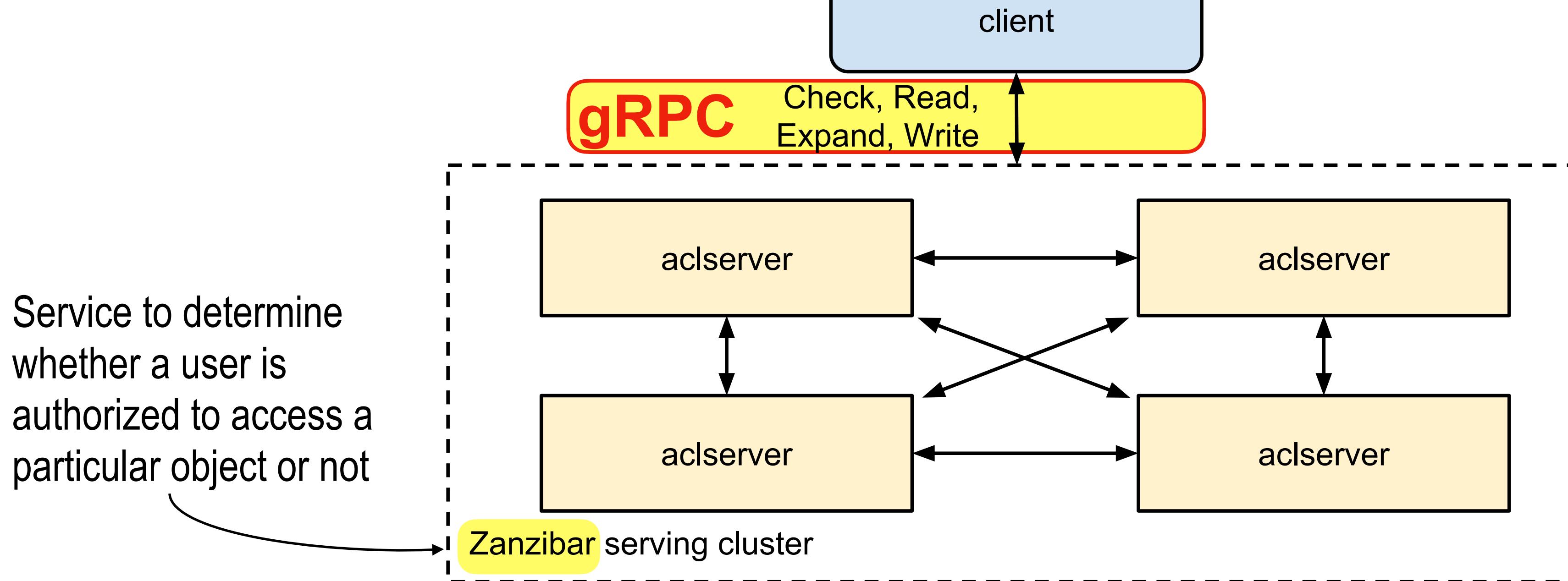
# Examples of RPC systems

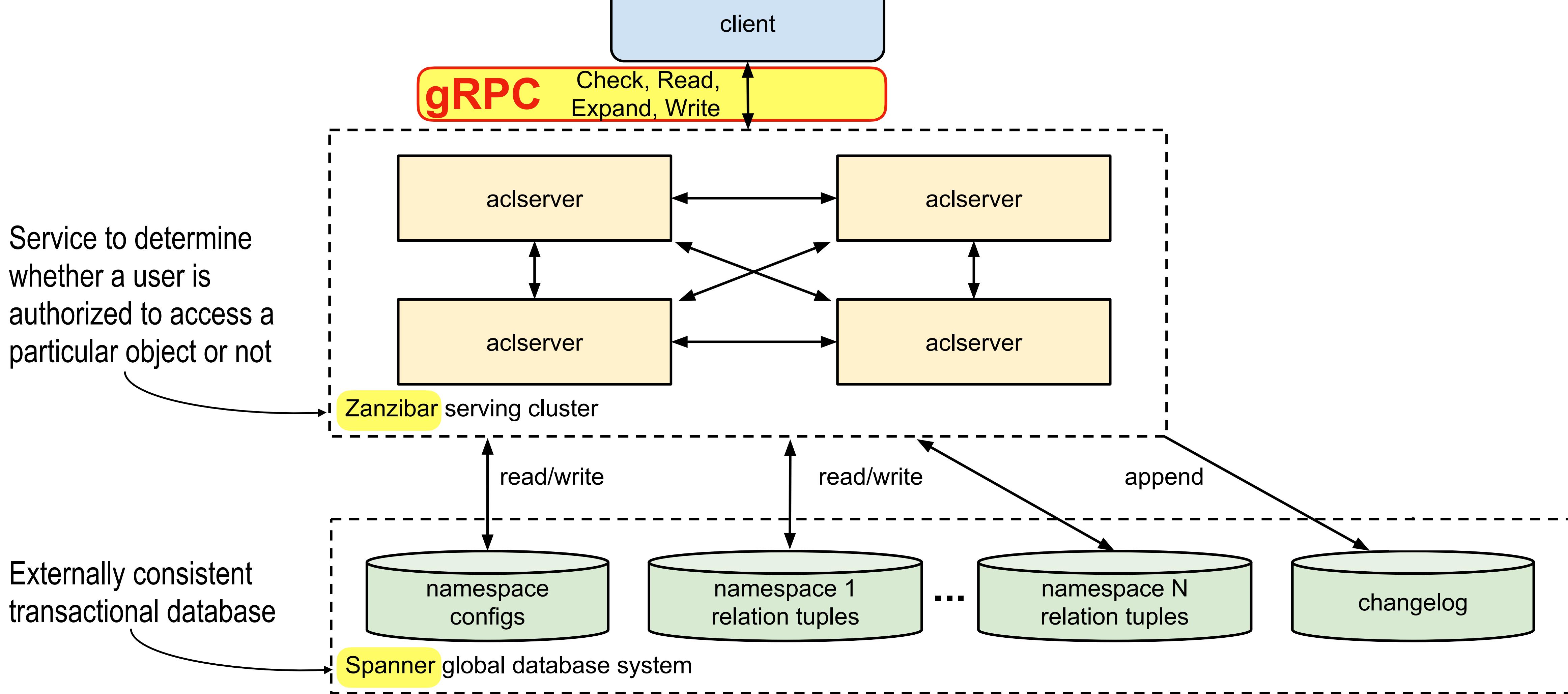
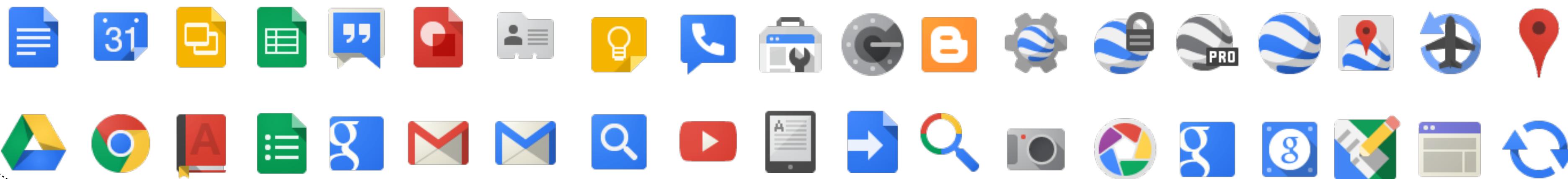
- NFS and  Google Cloud Filestore
- Java RMI and Google Web Toolkit
- Go's rpc package
- Cassandra, HBase, Couchbase
- Apache Thrift
- gRPC (uses Google Protocol Buffers IDL)
- *microservices, mobile, real-time, IoT, ...*

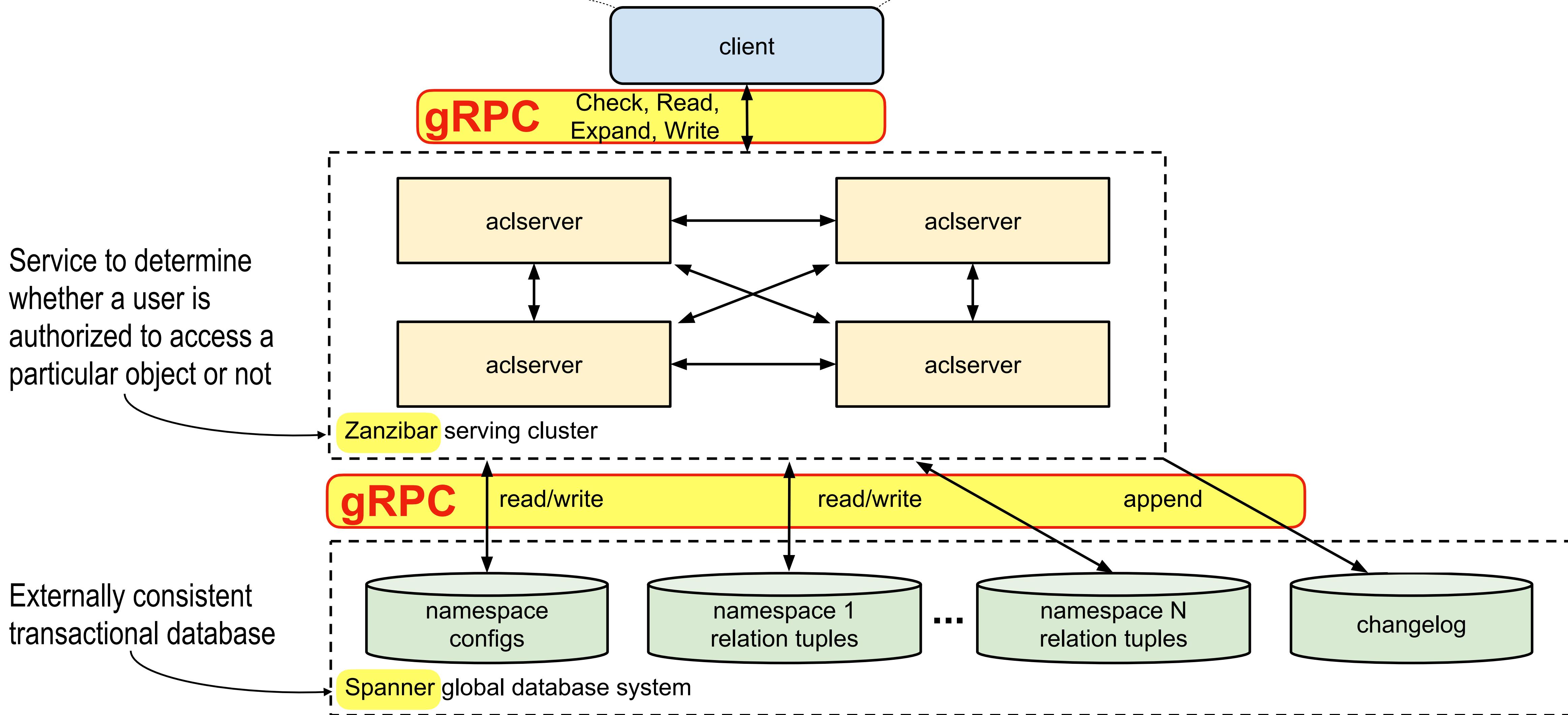
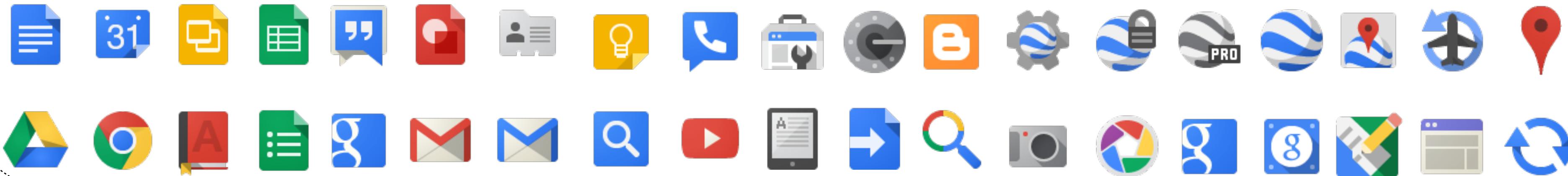






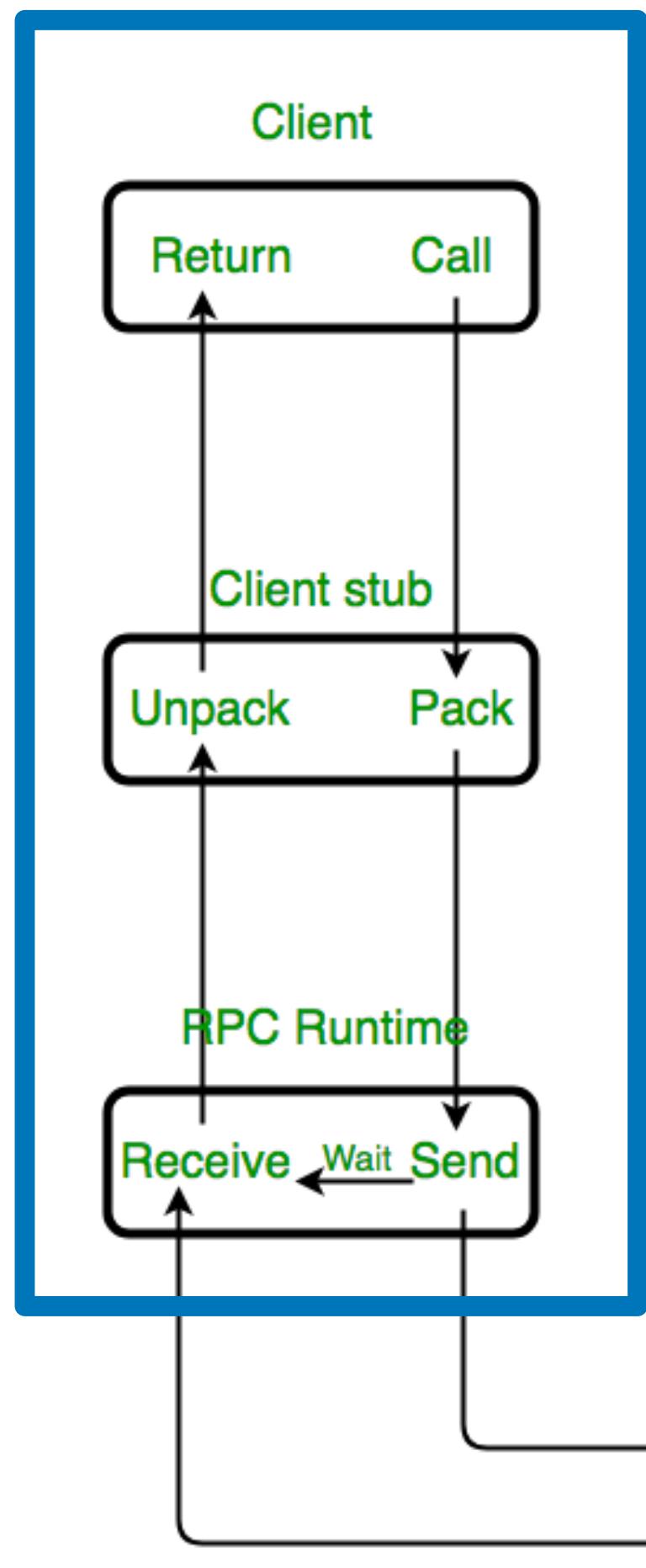




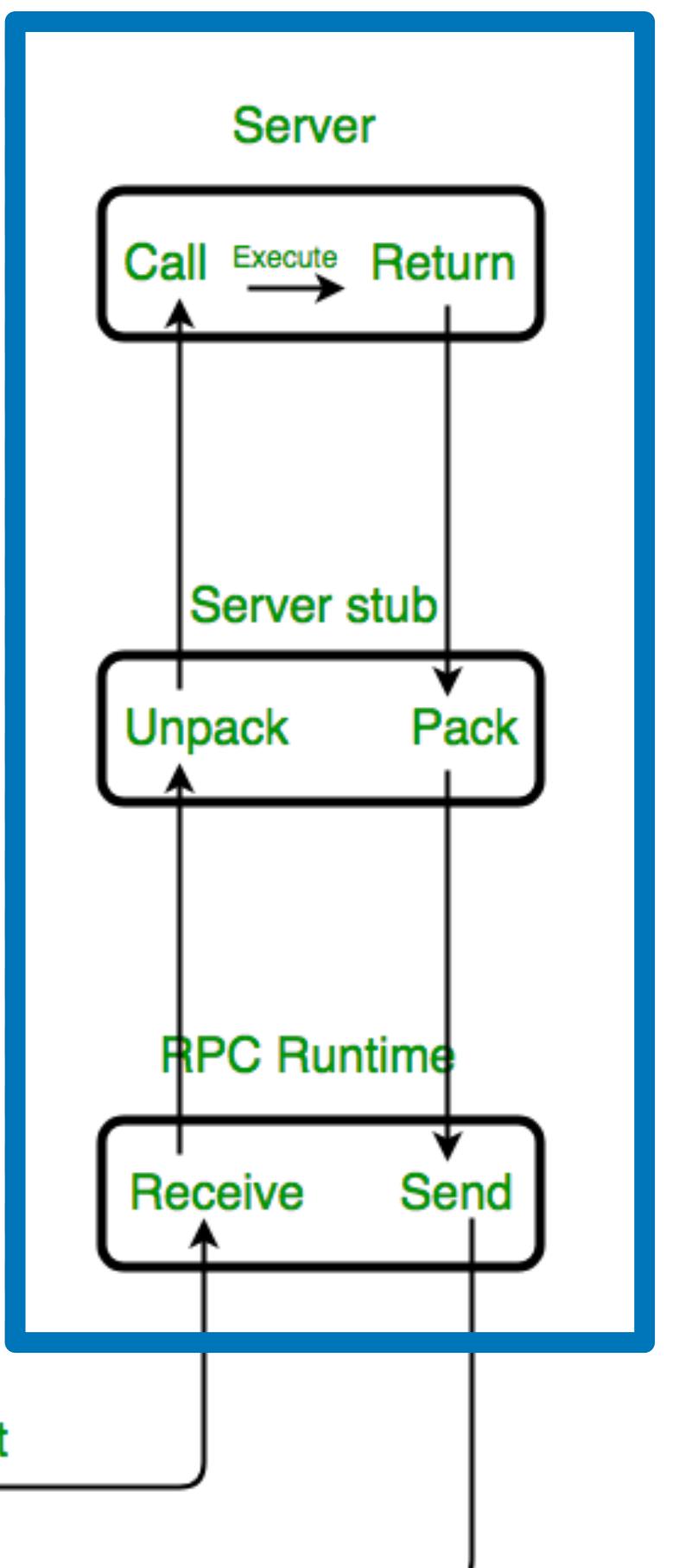


# Workflow for writing RPC-based systems

*Client address space*

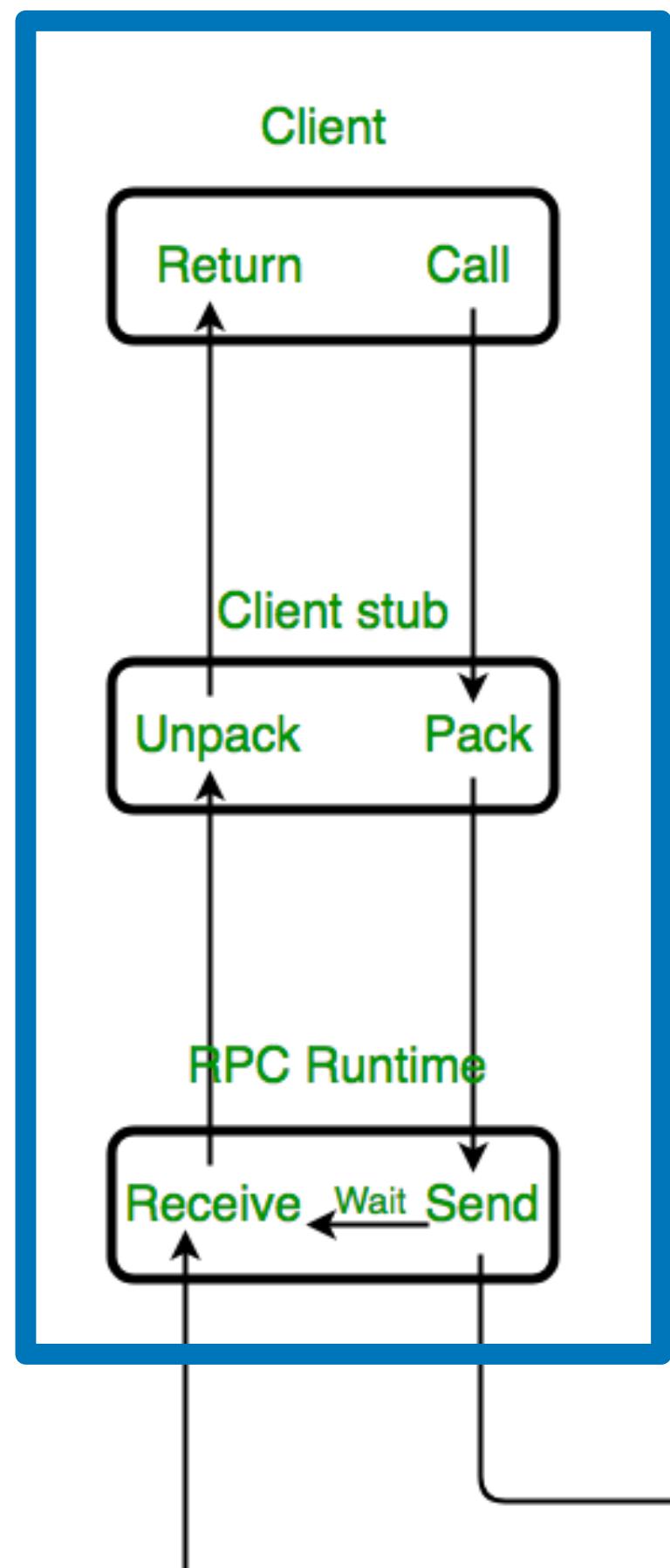


*Server address space*

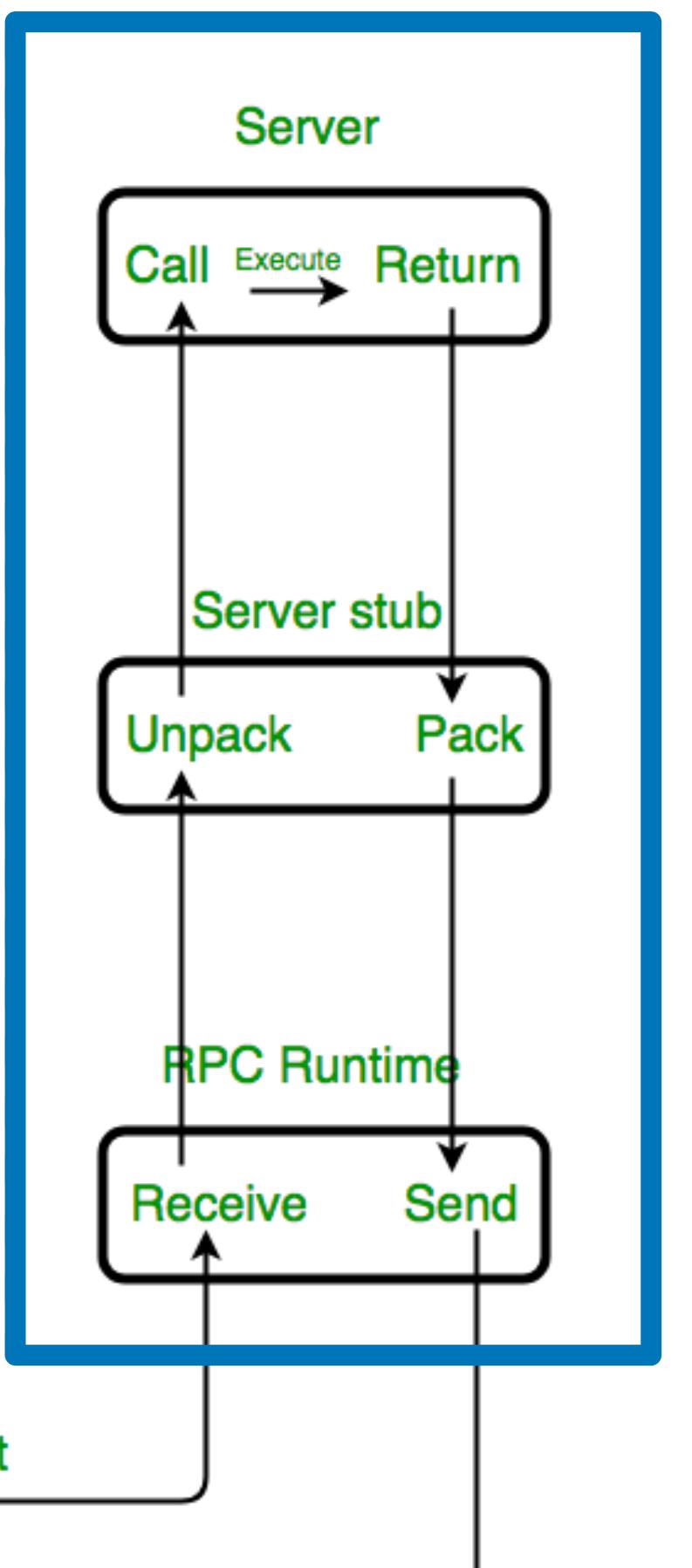


# Workflow for writing RPC-based systems

*Client address space*

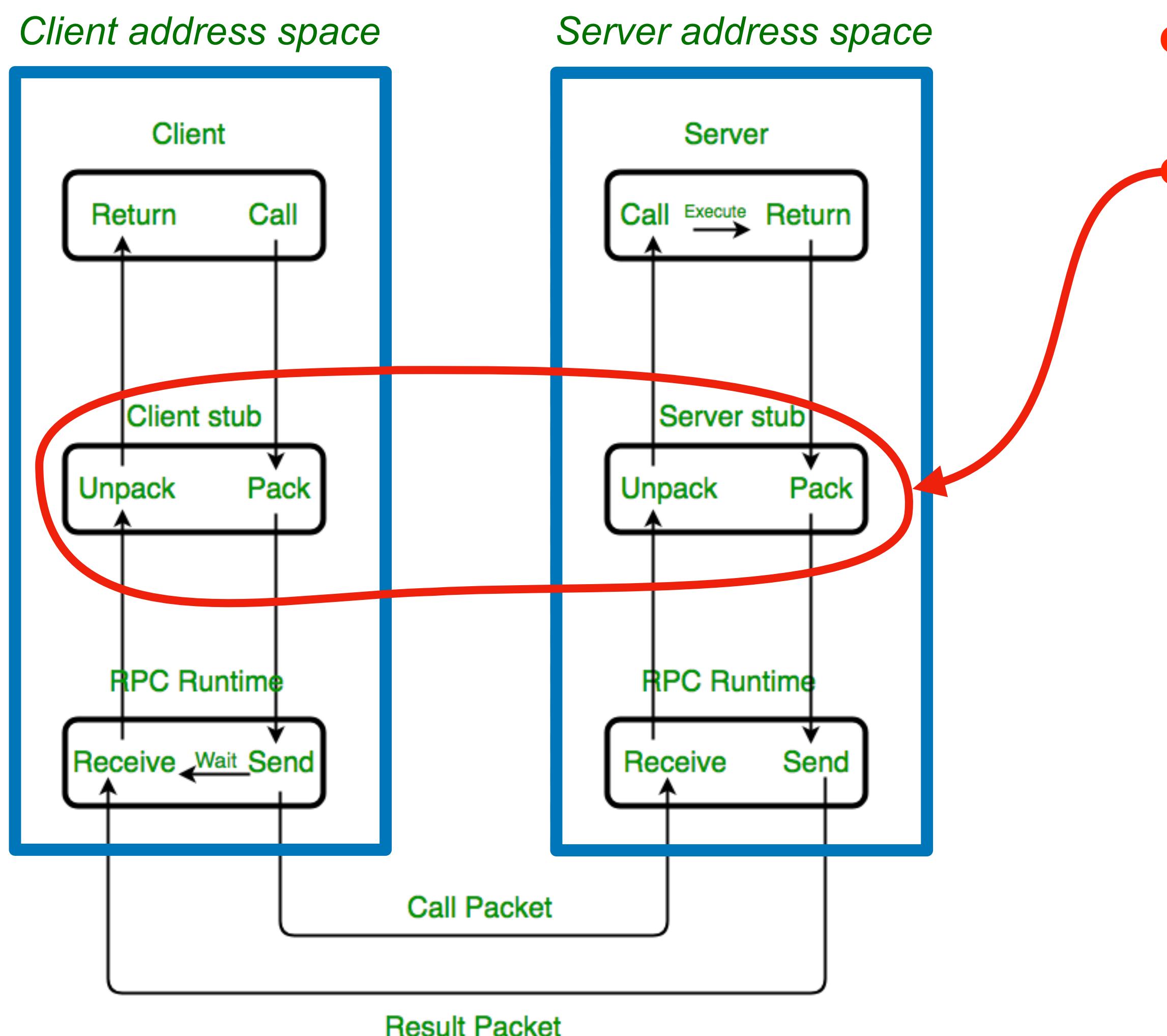


*Server address space*



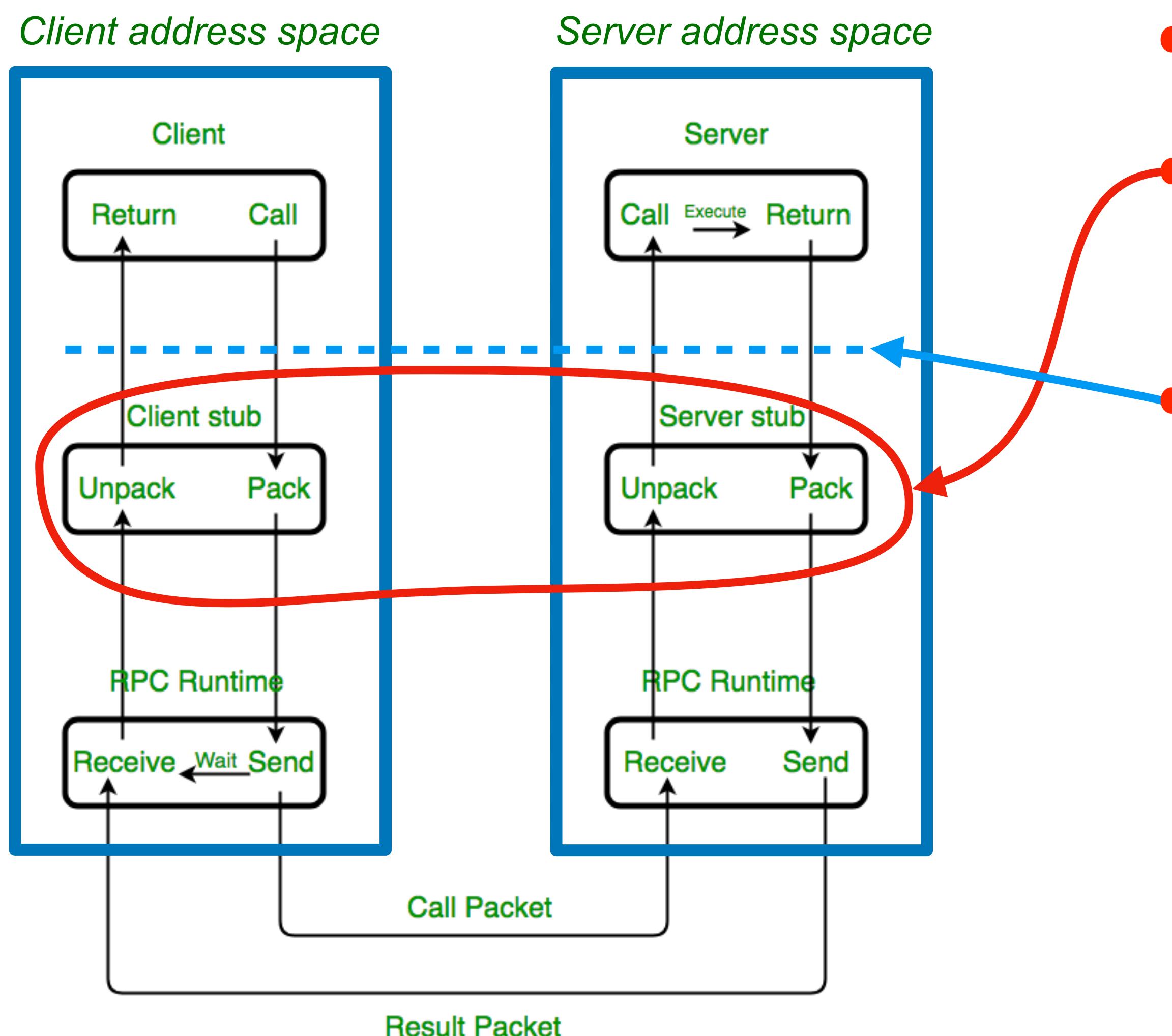
- Define the service in an IDL file

# Workflow for writing RPC-based systems



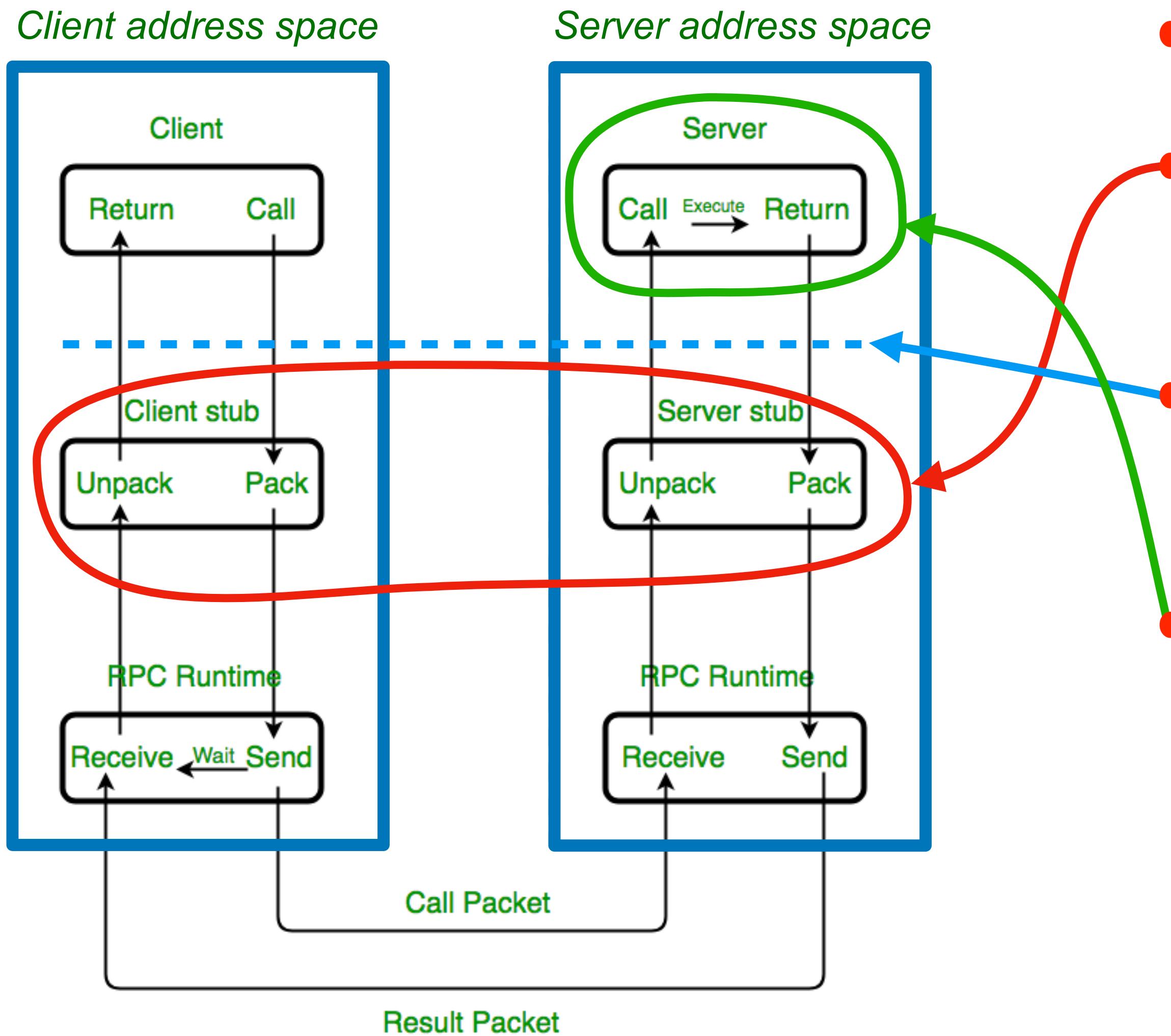
- Define the service in an IDL file  
Generate message implementations using the IDL compiler

# Workflow for writing RPC-based systems



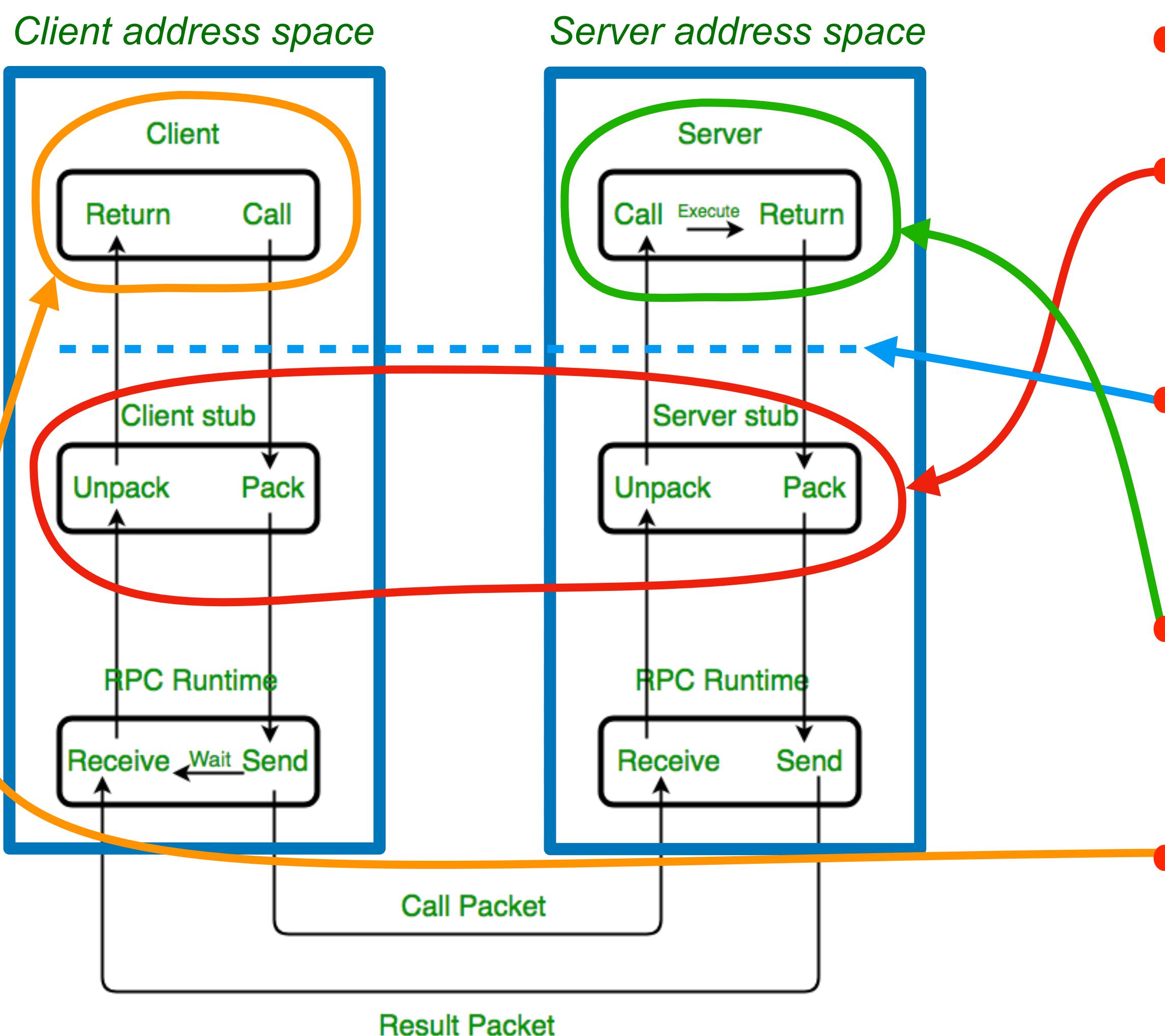
- Define the service in an IDL file
- Generate message implementations using the IDL compiler
- Generate server and client code using the RPC compiler

# Workflow for writing RPC-based systems



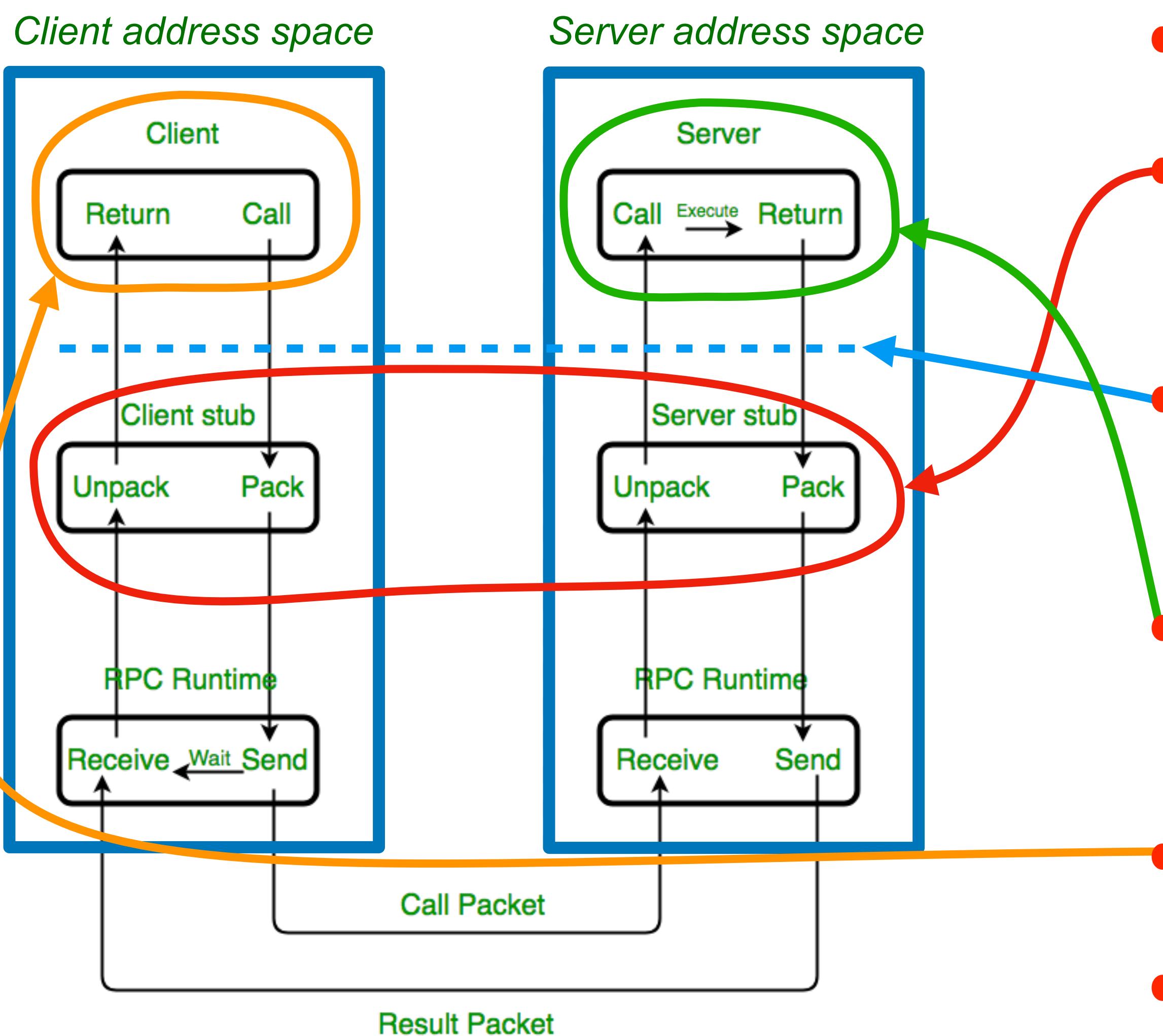
- Define the service in an IDL file
- Generate message implementations using the IDL compiler
- Generate server and client code using the RPC compiler
- Write the server to implement the generated interface

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

## Methods

<a href="#">BatchCreateSessions</a>	Creates multiple new sessions.
<a href="#">BatchWrite</a>	Batches the supplied mutation groups in a collection of efficient transactions.
<a href="#">BeginTransaction</a>	Begins a new transaction.
<a href="#">Commit</a>	Commits a transaction.
<a href="#">CreateSession</a>	Creates a new session.
<a href="#">DeleteSession</a>	Ends a session.
<a href="#">ExecuteBatchDml</a>	Executes a batch of DML statements.
<a href="#">ExecuteSql</a>	Executes an SQL statement, returning all results in a single reply.  This method is annotated with <b>rpc ExecuteSql(ExecuteSqlRequest) returns (ResultSet)</b> .  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 <b>FAILED_PRECONDITION</b> error.  Operations inside read-write transactions might return <b>ABORTED</b> . If this occurs, the application should restart the transaction from the beginning. See <a href="#">Transaction</a> for more details.  Larger result sets can be fetched in streaming fashion by calling <a href="#">ExecuteStreamingSql</a> instead.
<a href="#">ExecuteStreamingSql</a>	Like <a href="#">ExecuteSql</a> , except returns the result set as a stream.
<a href="#">GetSession</a>	Gets a session.
<a href="#">ListSessions</a>	Lists all sessions in a given database.
<a href="#">PartitionQuery</a>	Creates a set of partition tokens that can be used to execute a query operation in parallel.
<a href="#">PartitionRead</a>	Creates a set of partition tokens that can be used to execute a read operation in parallel.
<a href="#">Read</a>	Reads rows from the database using key lookups and scans, as a simple key/value style alternative to <a href="#">ExecuteSql</a> .
<a href="#">Rollback</a>	Rolls back a transaction, releasing any locks it holds.
<a href="#">StreamingRead</a>	Like <a href="#">Read</a> , except returns the result set as a stream.

## Methods

<b>BatchCreateSessions</b>	Creates multiple new sessions.
<b>BatchWrite</b>	Batches the supplied mutation groups in a collection of efficient transactions.
<b>BeginTransaction</b>	Begins a new transaction.
<b>Commit</b>	Commits a transaction.
<b>CreateSession</b>	Creates a new session.
<b>DeleteSession</b>	Ends a session.
<b>ExecuteBatchDml</b>	Executes a batch of DML statements.
<b>ExecuteSql</b>	Executes an SQL statement, returning all results in a single reply.
<b>ExecuteStreamingSql</b>	Like <b>ExecuteSql</b> , except returns the result set as a stream.
<b>GetSession</b>	Gets a session.
<b>ListSessions</b>	Lists all sessions in a given database.
<b>PartitionQuery</b>	Creates a set of partition tokens that can be used to execute queries.
<b>PartitionRead</b>	Creates a set of partition tokens that can be used to execute reads.
<b>Read</b>	Reads rows from the database using key lookups and returns them to <b>ExecuteSql</b> .
<b>Rollback</b>	Rolls back a transaction, releasing any locks it holds.
<b>StreamingRead</b>	Like <b>Read</b> , 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";
    }
}
```

## Methods

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

---

# Benefits of RPC

---

- Strong modularity with the convenience of a procedure call

# 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

---

# Drawbacks of RPC

---

---

- RPCs typically take longer than a local procedure call
  - *Leaky abstraction*

# 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?*
  - ... ?

# 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

# RPC Semantics

---

---

- At-least-once semantics     $\geq 1$  execution

# RPC Semantics

---

- At-least-once semantics     $\geq 1$  execution
- At-most-once semantics     $\leq 1$  executions

# RPC Semantics

---

---

- At-least-once semantics     $\geq 1$  execution
- At-most-once semantics     $\leq 1$  executions
- Exactly-once semantics     $= 1$  execution

# **How to implement exactly-once RPC semantics ?**

# **How to implement exactly-once RPC semantics ?**

# RPC Tax in Data Centers

# RPC Tax in Data Centers

---

- Network latency and transmission overhead

# RPC Tax in Data Centers

---

- Network latency and transmission overhead
- Context switching on both client and server

# RPC Tax in Data Centers

---

- Network latency and transmission overhead
- Context switching on both client and server
- Memory and serialization overheads

# RPC Tax in Data Centers

---

- Network latency and transmission overhead
- Context switching on both client and server
- Memory and serialization overheads
- Network load

# 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

# 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
- ...

# 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

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

*same address space*

Memory safety

Message-based communication

*separate address spaces*



**10m15s**

## **Memory Safety**

*Fundamental requirement for  
good modularity within the same address space*

# Memory Safety

---

---

Based on Nagarakatte et al., [SoftBound: Highly Compatible and Complete Spatial Memory Safety for C](#), PLDI 2009

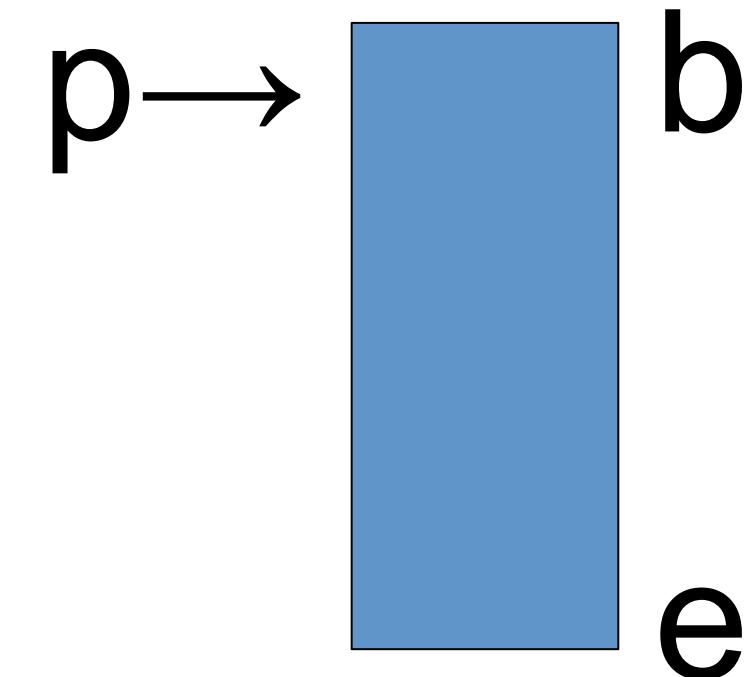
# Memory Safety

---

- Memory can be defined (allocated) or undefined (not allocated)
  - *Deallocated memory cannot be reused prior to (re)allocation*

# Memory Safety

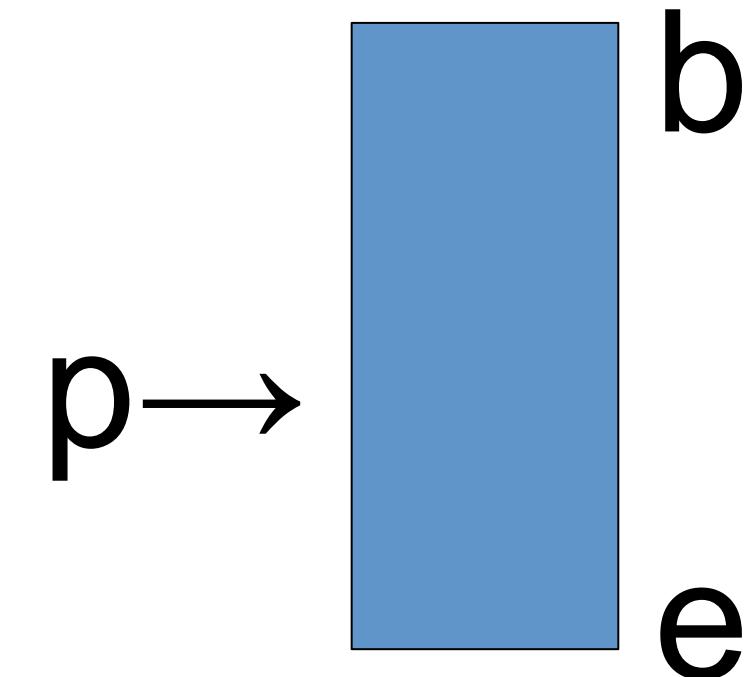
- Memory can be defined (allocated) or undefined (not allocated)
  - *Deallocated memory cannot be reused prior to (re)allocation*
- Pointer is a capability ( $p, b, e$ )
  - *Base  $b$ , extent  $e$ , pointer  $p$*



Based on Nagarakatte et al., [SoftBound: Highly Compatible and Complete Spatial Memory Safety for C](#), PLDI 2009

# Memory Safety

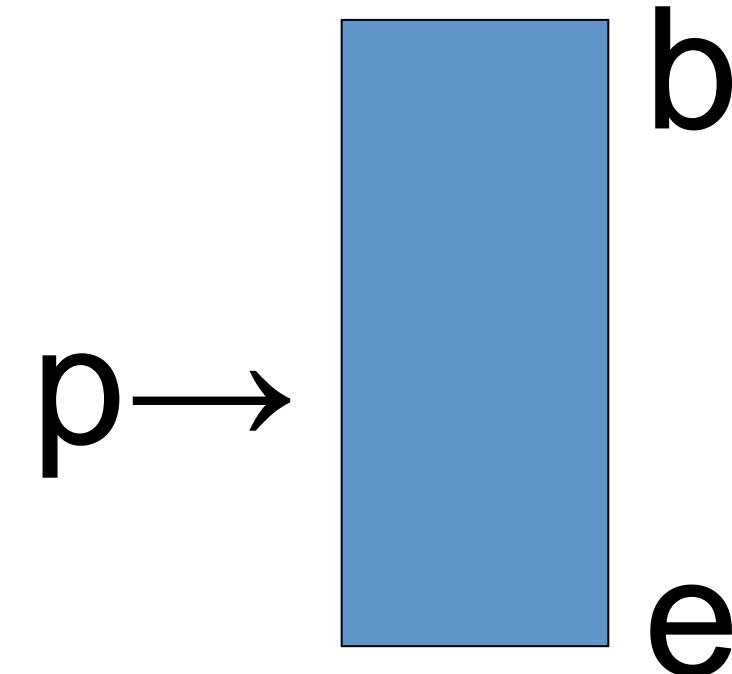
- Memory can be defined (allocated) or undefined (not allocated)
  - *Deallocated memory cannot be reused prior to (re)allocation*
- Pointer is a capability ( $p, b, e$ )
  - *Base  $b$ , extent  $e$ , pointer  $p$*



Based on Nagarakatte et al., [SoftBound: Highly Compatible and Complete Spatial Memory Safety for C](#), PLDI 2009

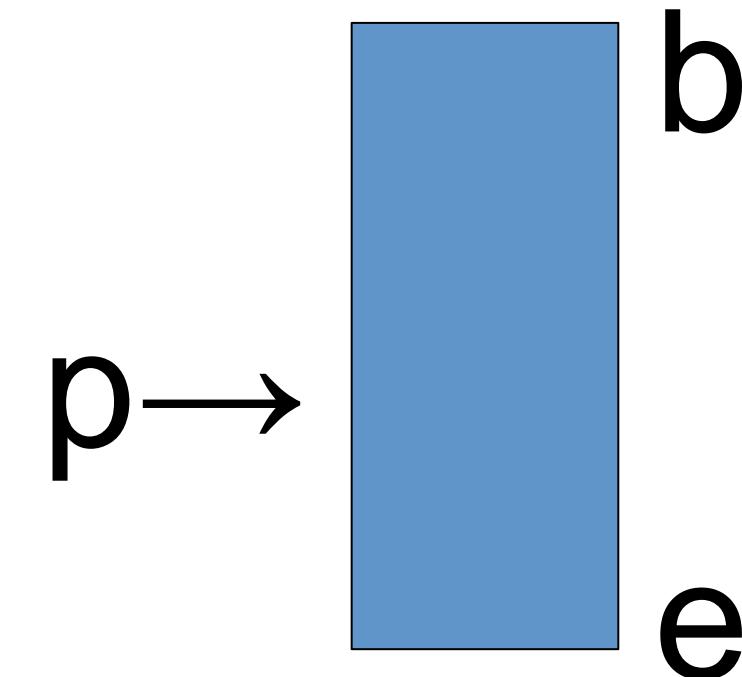
# Memory Safety

- 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



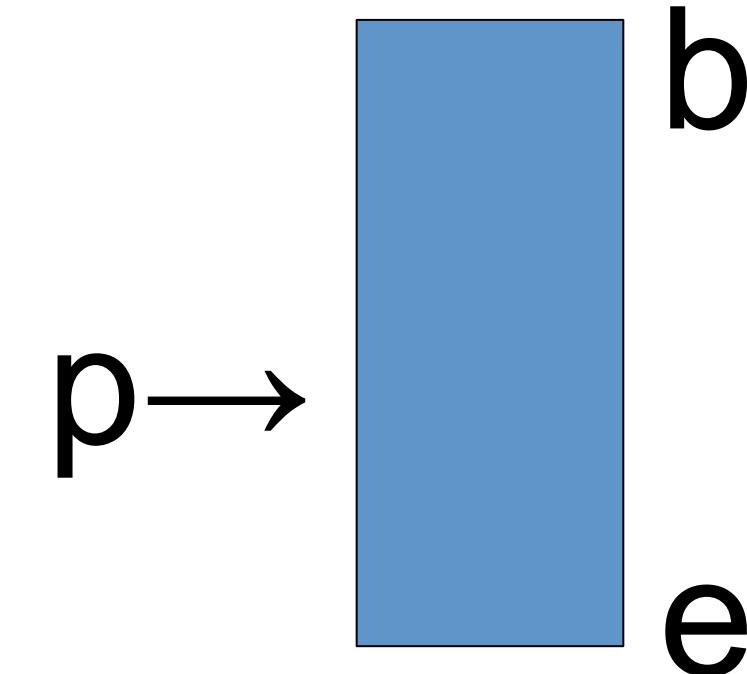
# Memory Safety

- 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  $\Leftrightarrow$  all ptr derefs in that exec are safe



# Memory Safety

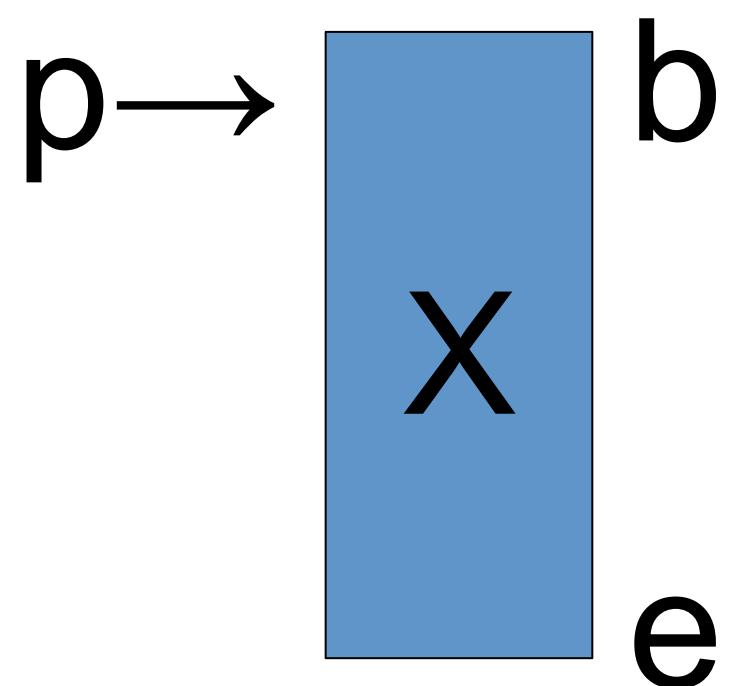
- Memory can be defined (allocated) or undefined (not allocated)
  - *Deallocated memory cannot be reused prior to (re)allocation*
- Pointer is a capability ( $p, b, e$ )
  - *Base  $b$ , extent  $e$ , pointer  $p$*
- $*p$  is safe iff it accesses memory within the target obj that  $p$  is based on
- An execution is memory-safe  $\Leftrightarrow$  all ptr derefs in that exec are safe
- A program is memory-safe  $\Leftrightarrow$  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

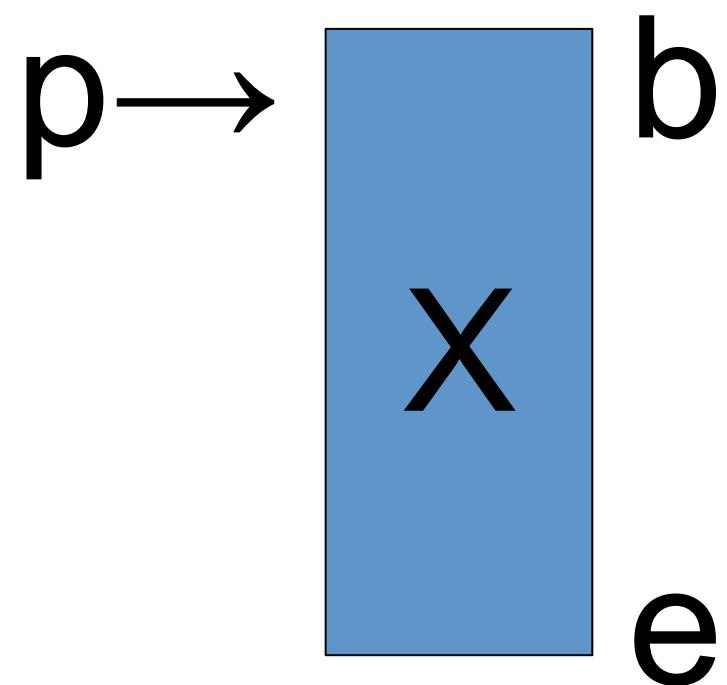
# “Based on” relationship

- p is based on memory object X iff p is



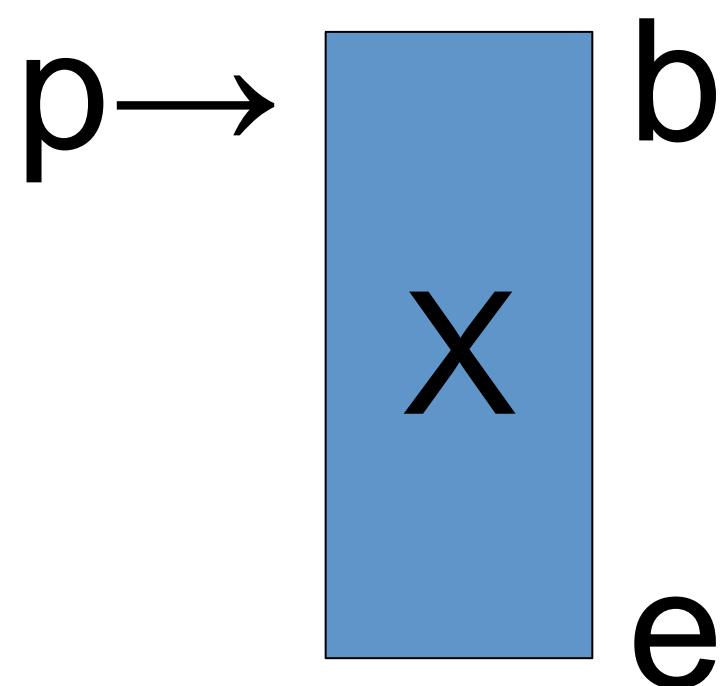
# “Based on” relationship

- p is based on memory object X iff p is
  1. obtained by allocating X at runtime on the heap, or



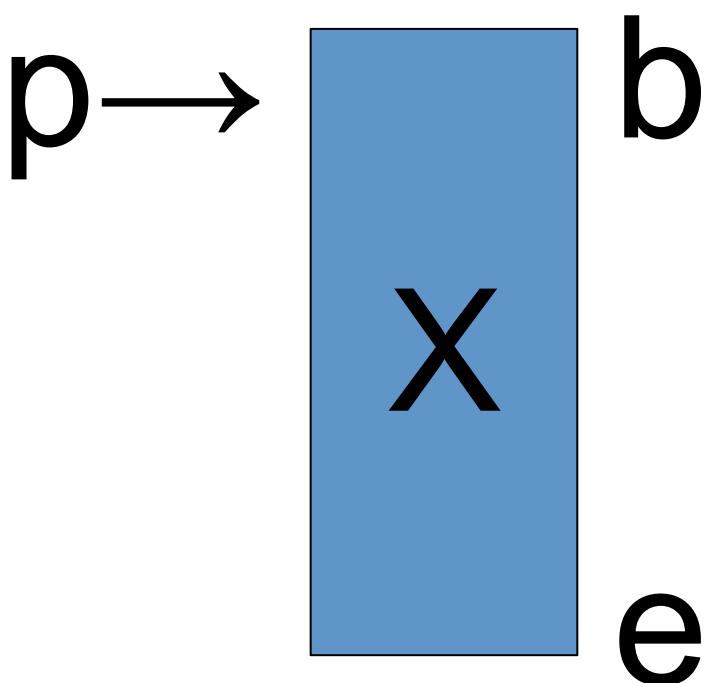
# “Based on” relationship

- 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



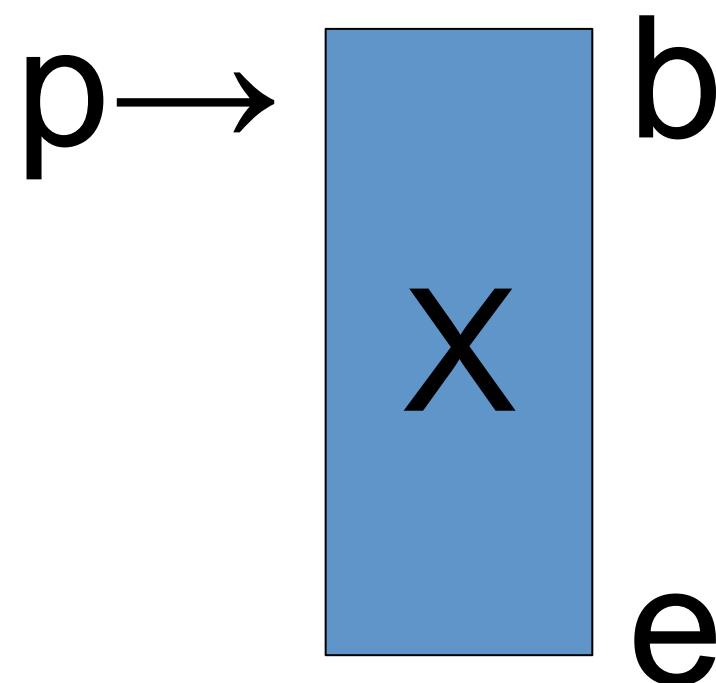
# “Based on” relationship

- 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*



# “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<sup>1</sup>
$$b \leq p \leq e$$

<sup>1</sup> and that memory is defined

# 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<sup>1</sup>
$$b \leq p \leq e$$
- An execution is memory-safe  $\Leftrightarrow$   
all pointer dereferences in that execution are safe

<sup>1</sup> and that memory is defined

# 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<sup>1</sup>
$$b \leq p \leq e$$
- An execution is memory-safe  $\Leftrightarrow$   
all pointer dereferences in that execution are safe
- A program is memory-safe  $\Leftrightarrow$   
all possible executions (for all possible inputs) are memory-safe

<sup>1</sup> and that memory is defined

# Memory Safety

---

- Memory safety is fundamental to in-memory client/server

# 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

# Memory Safety

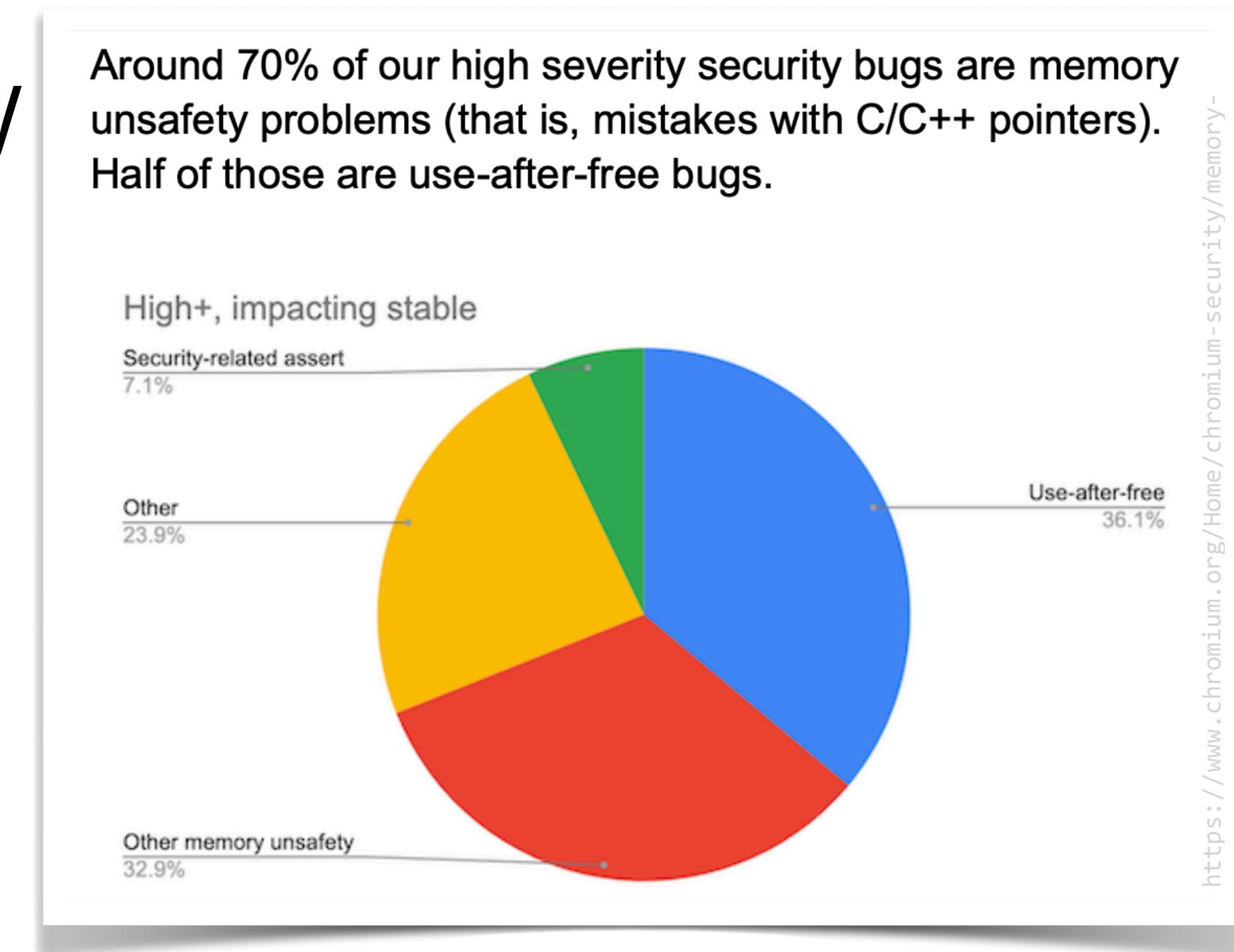
---

- 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

# 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.



# Outline

- Recap of modularization
  - Local procedure calls (module = procedure)
  - Program objects & types (module = memory objects)
  - Client/server architecture (different address spaces)
  - Example: Remote procedure calls
- 
- Memory safety
- Message-based communication

