EPFL

The Client/Server Design Pattern

Prof. George Candea
School of Computer & Communication Sciences

Outline

- Recap of modularization

same address space

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

Memory safety

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

Message-based communication

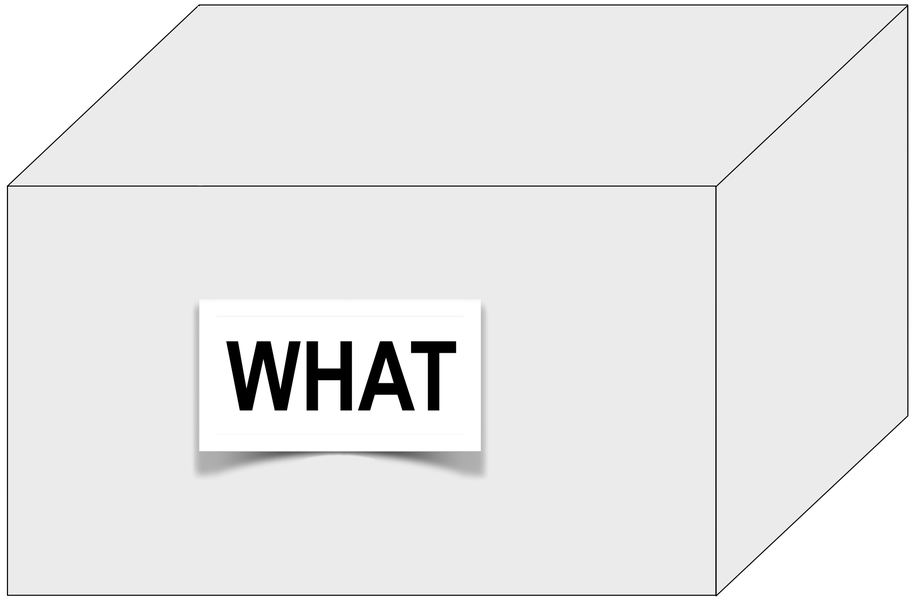
separate address spaces

Modularity

Modularity

Abstraction and Interfaces

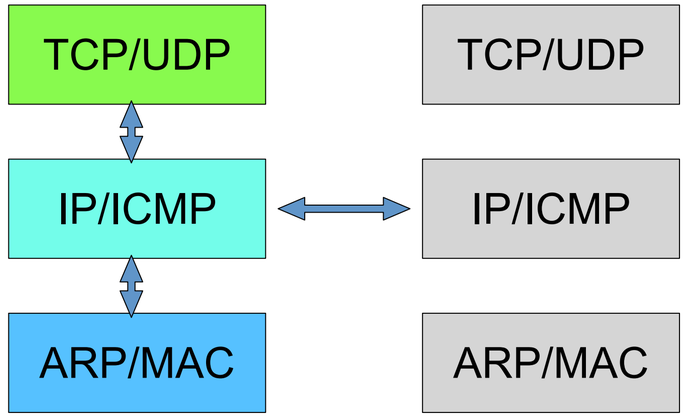
- Specify “what” a component/subsystem does
- Together with modularity,
  separates “what” from “how”
  => abstraction



Names

- Scope
  - *Private: unique within a context (e.g., a private IP address)*
  - *Global: unique across contexts (e.g., a global IP address)*
- Structure
  - *Hierarchical: name relationship implies object relationship (e.g., two IP addresses sharing the same prefix)*
  - *Flat: name relationship implies nothing (e.g., content IDs in Peer-to-Peer networks)*
- Naming system
  - *Directories of name->value mappings, support name lookups and updates*

Layers



- Layer = group of modules
  - *Internet transport layer = UDP + TCP*
  - *Internet network layer = IP*
- Module communicates with modules in layer above/below, on the same layer in different stack instances, through API
  - *send/receive calls/notifications*
- Module communicates with modules in the same layer stack, on a different stack instance, through a protocol

Outline

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

Memory safety

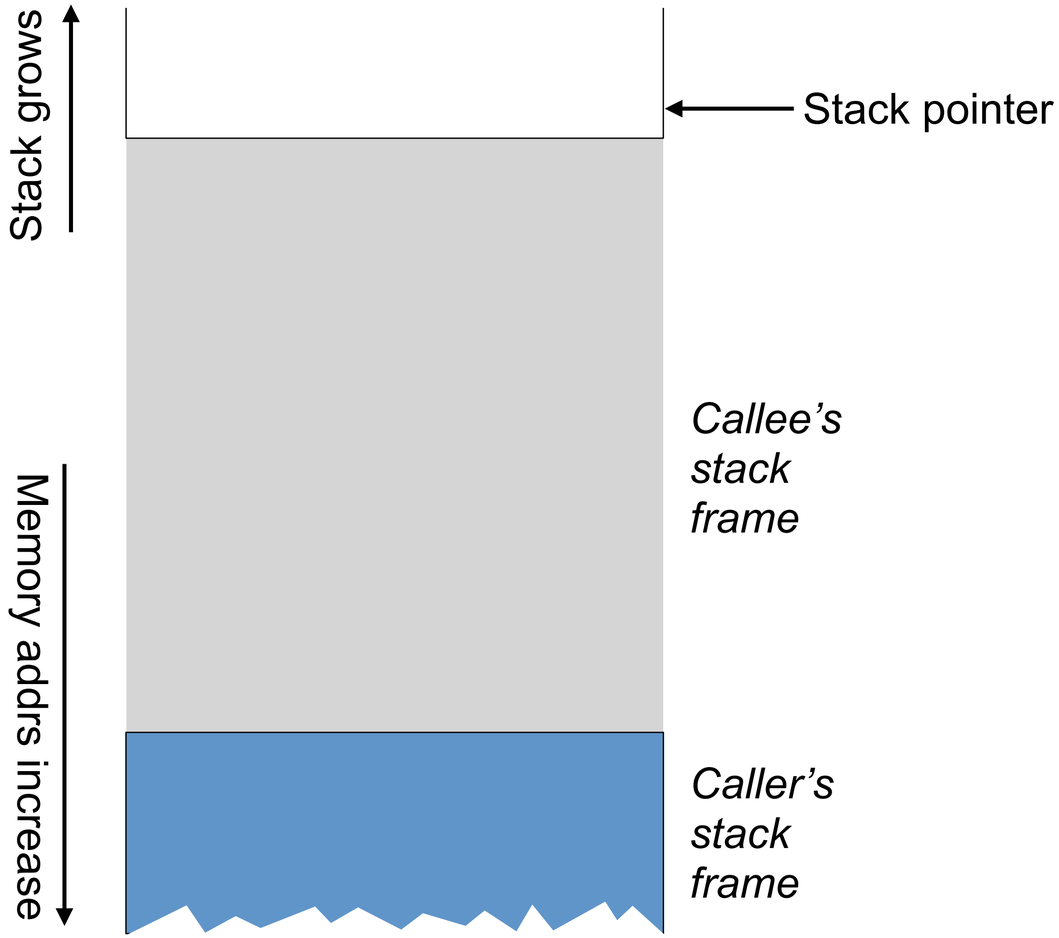
- Client/server architecture (different address spaces)
- Example: Remote procedure calls

Message-based communication

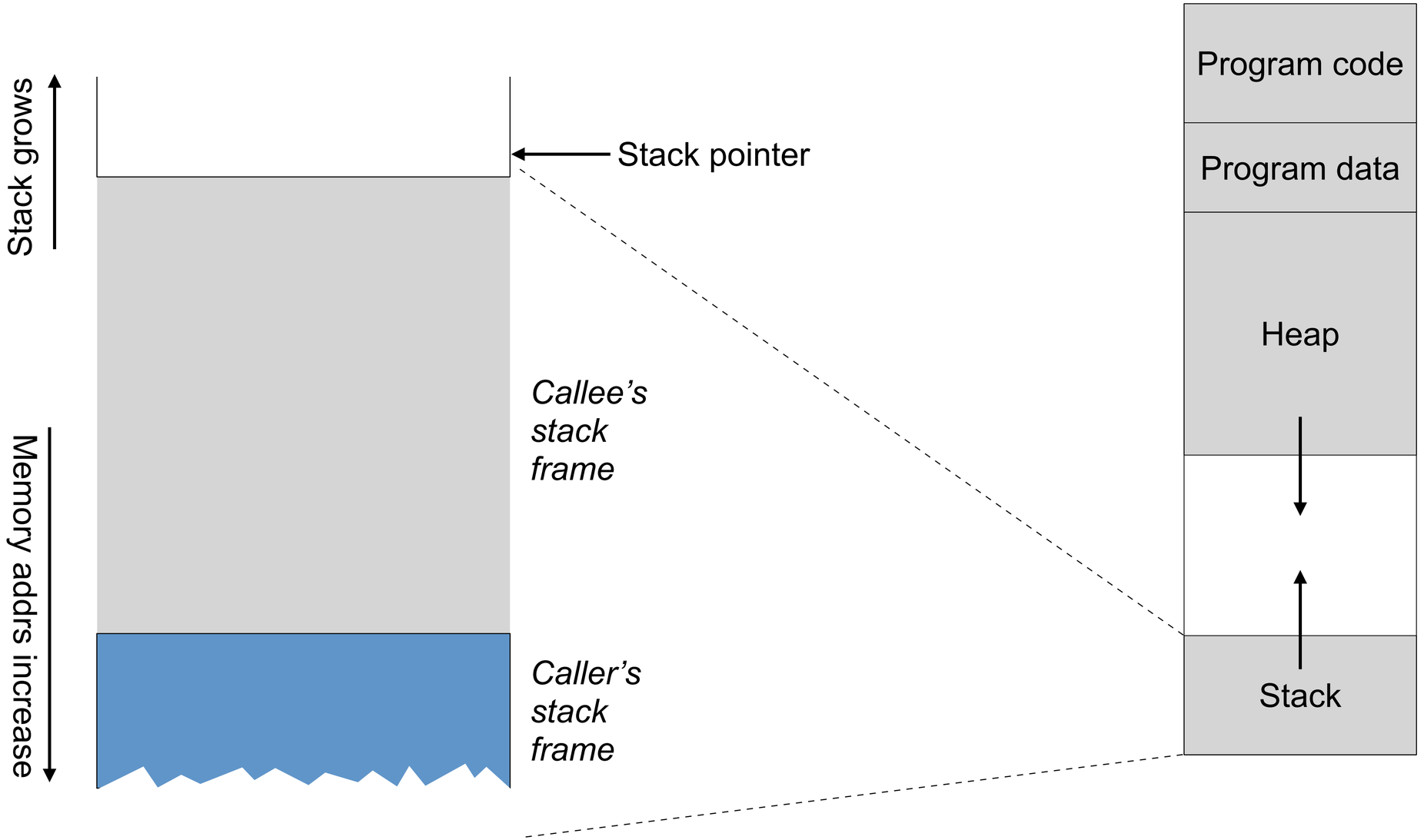
(Local) Procedure Calls

Basic mechanism for modularizing a program
(Modules = procedures)

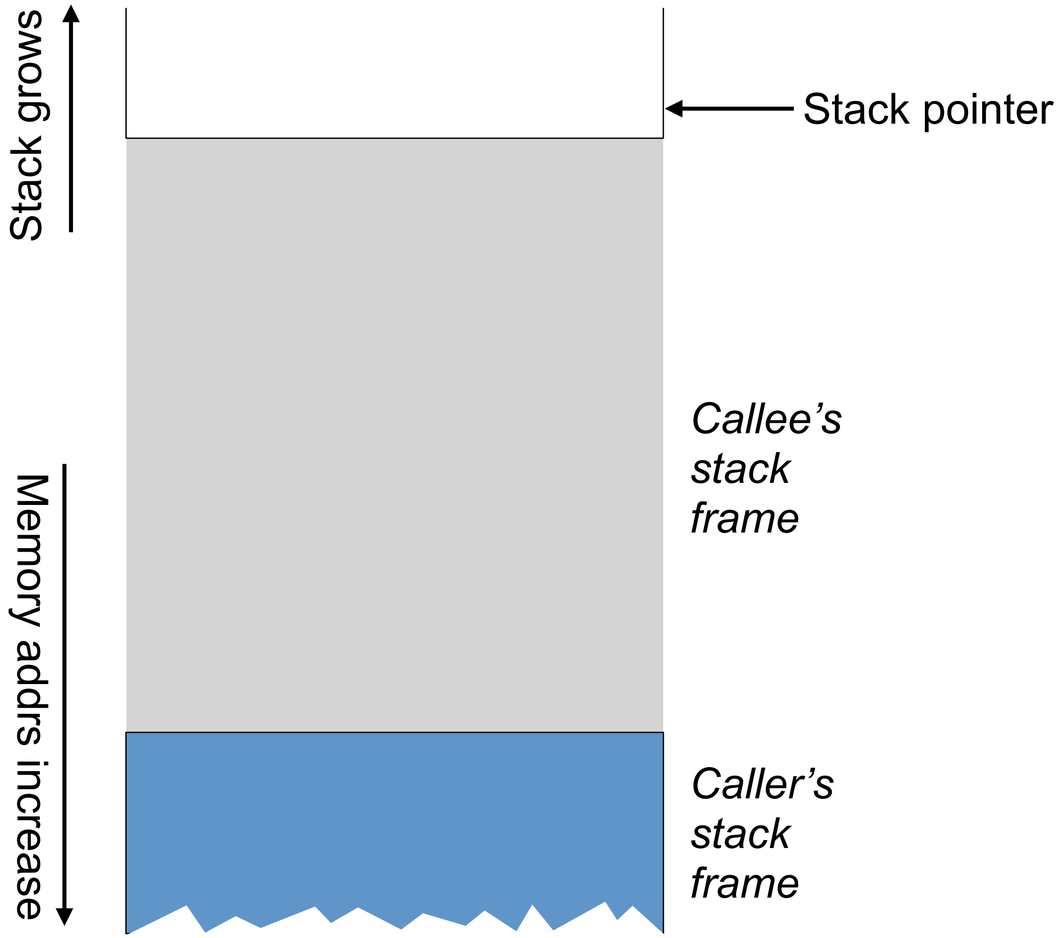
Stack-based calling convention



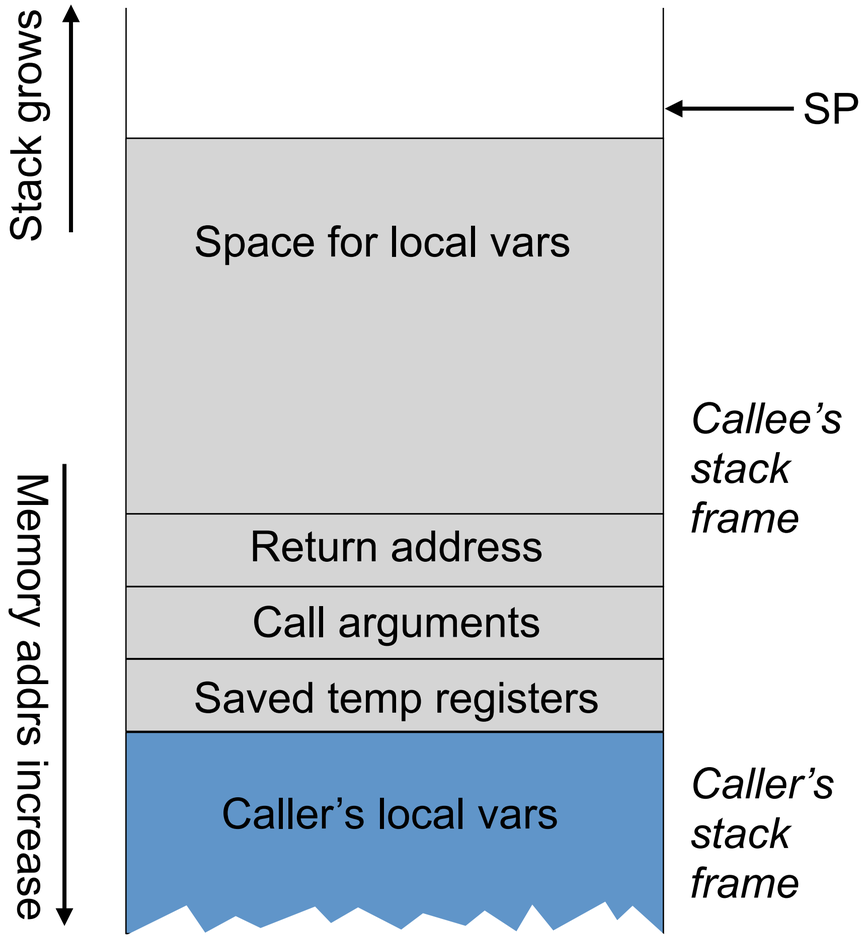
Stack-based calling convention



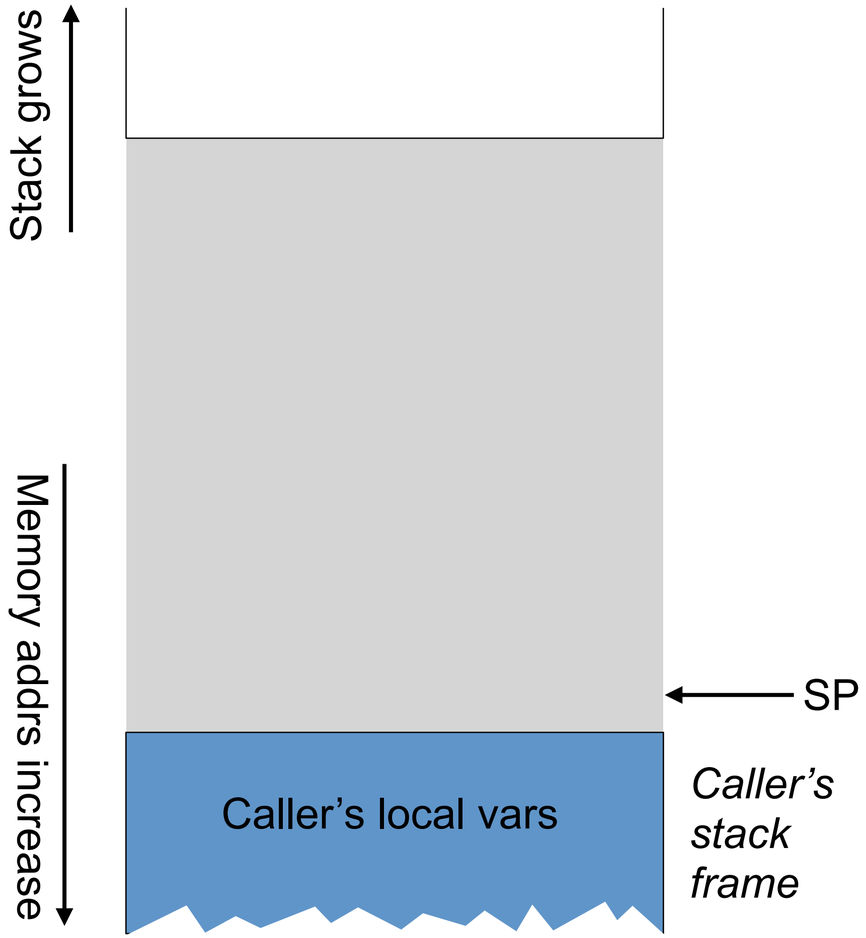
Stack-based calling convention



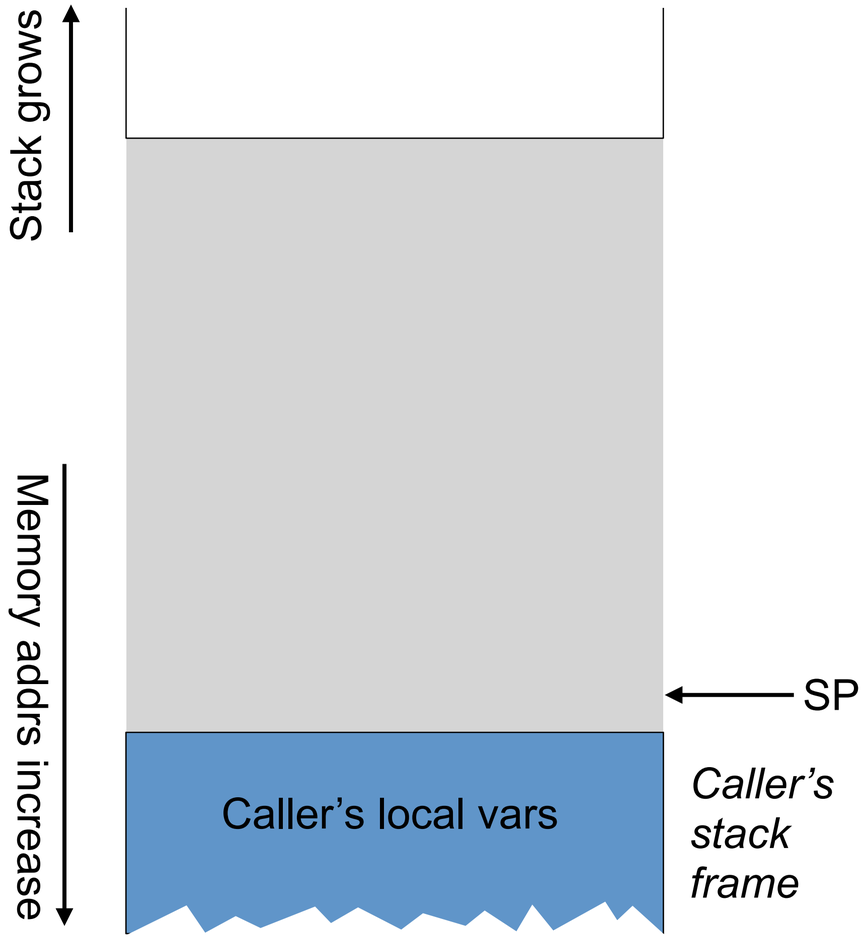
Stack-based calling convention



Stack-based calling convention



Stack-based calling convention



- ABI = interface between binary modules
- 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

- 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

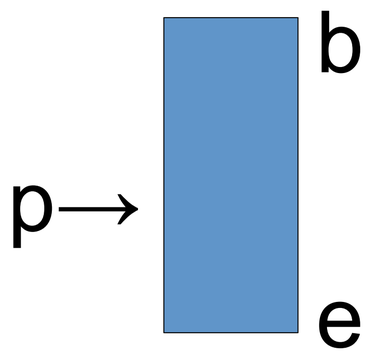
Message-based communication

Memory Safety

Fundamental requirement for
good modularity within the same address space

Memory Safety

- Memory can be defined (allocated) or undefined (not allocated)
  - *Assume deallocated memory is never reused*
- Pointer is a capability `(p,b,e)`
  - *Base **b**, extent **e**, pointer **p***



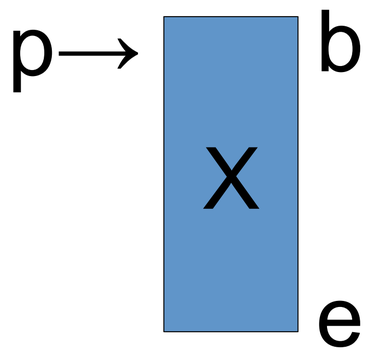
- `*p` is safe iff it accesses memory within the target obj that `p` is based on
- An execution is memory-safe <=> all ptr derefs in that exec are safe
- A program is memory-safe <=> all possible executions (for all possible inputs) are memory-safe

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
```

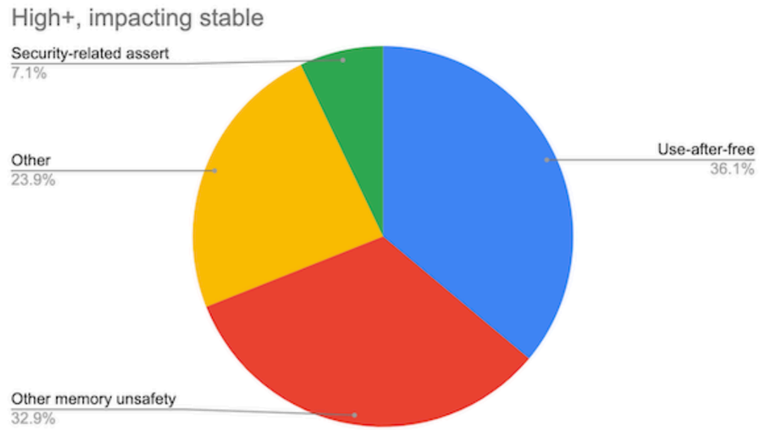
- An execution is memory-safe <=>
  all pointer dereferences in that execution are safe
- A program is memory-safe <=>
  all possible executions (for all possible inputs) are memory-safe

[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-

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

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

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

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

Data separate from Behavior

vs.

Encapsulation

```java
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

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

Memory safety

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

Message-based communication

Clients/Servers Interacting via Messages

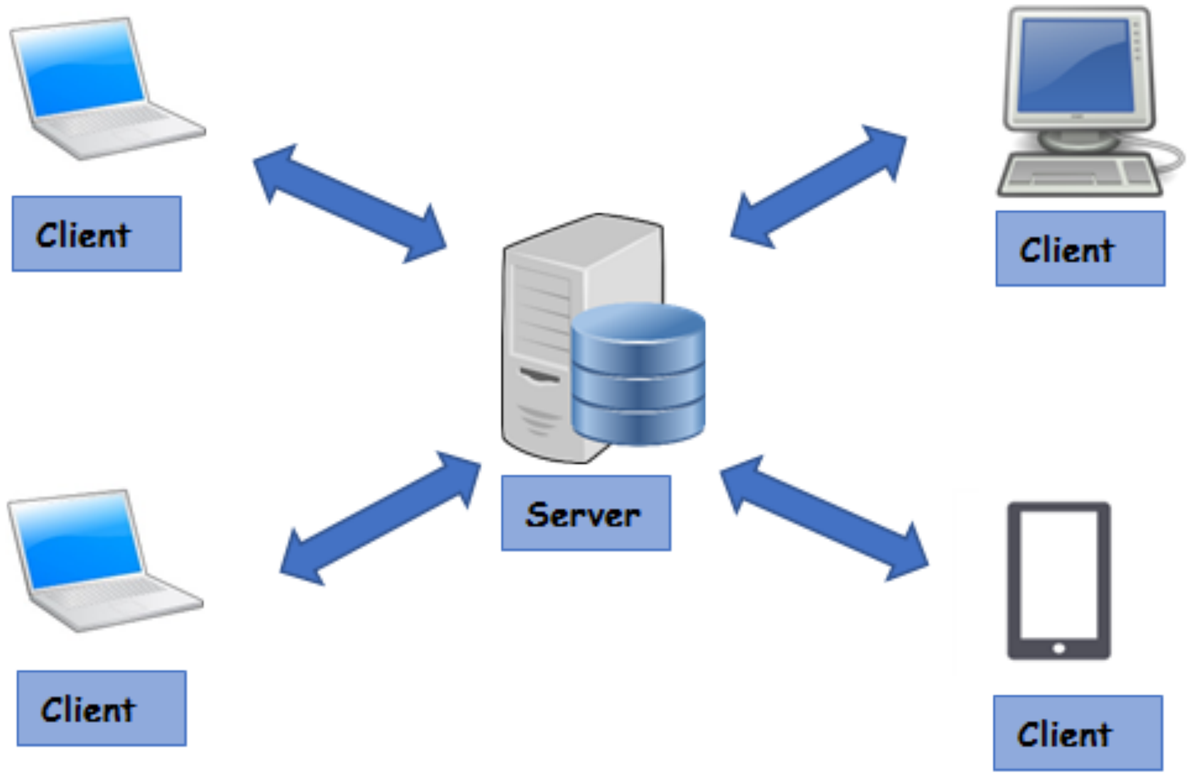
Modularization across different address spaces

Splitting into Clients and Servers

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

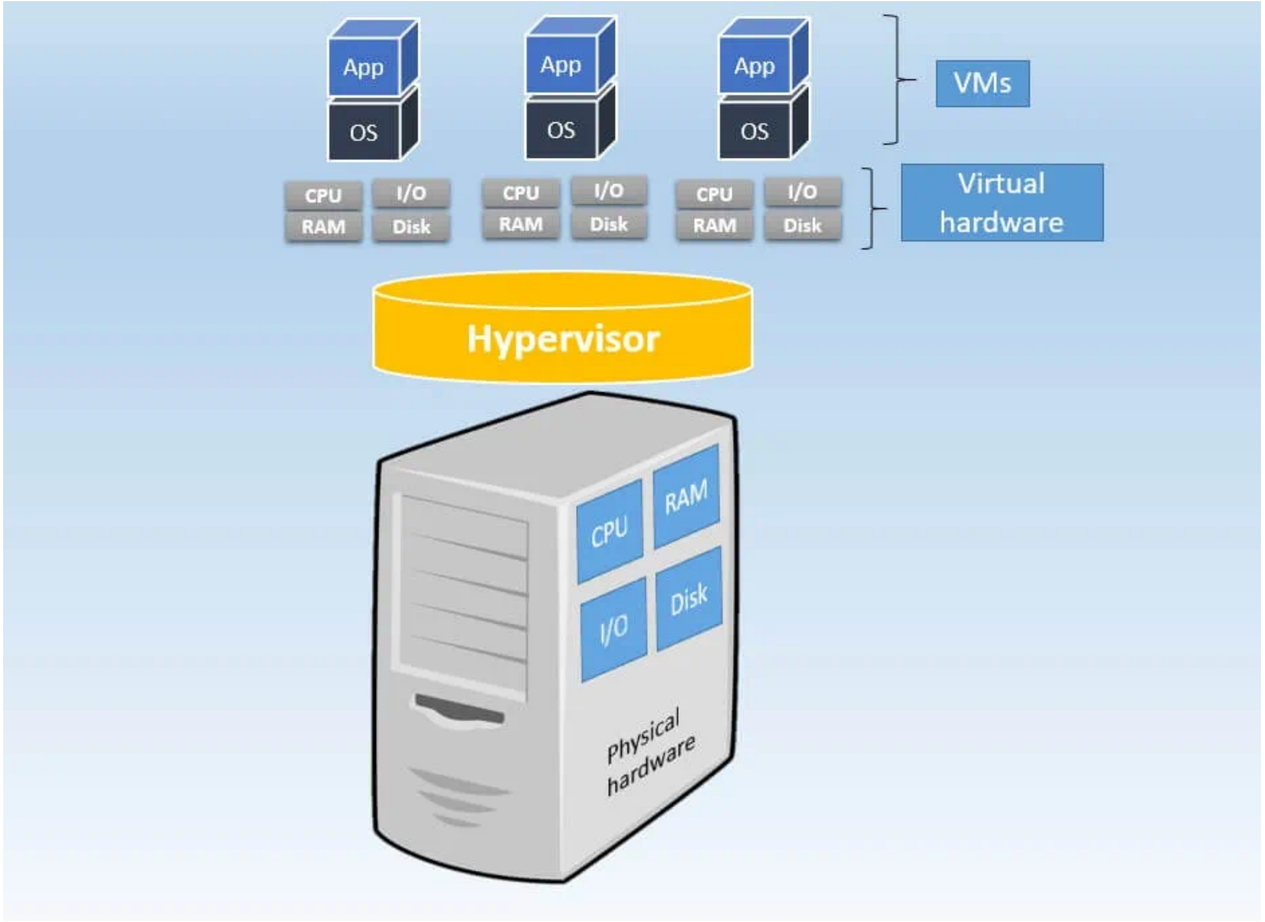
- Rely on physics
- Reduce fate sharing
- Improve encapsulation



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

Physical (and virtual) servers

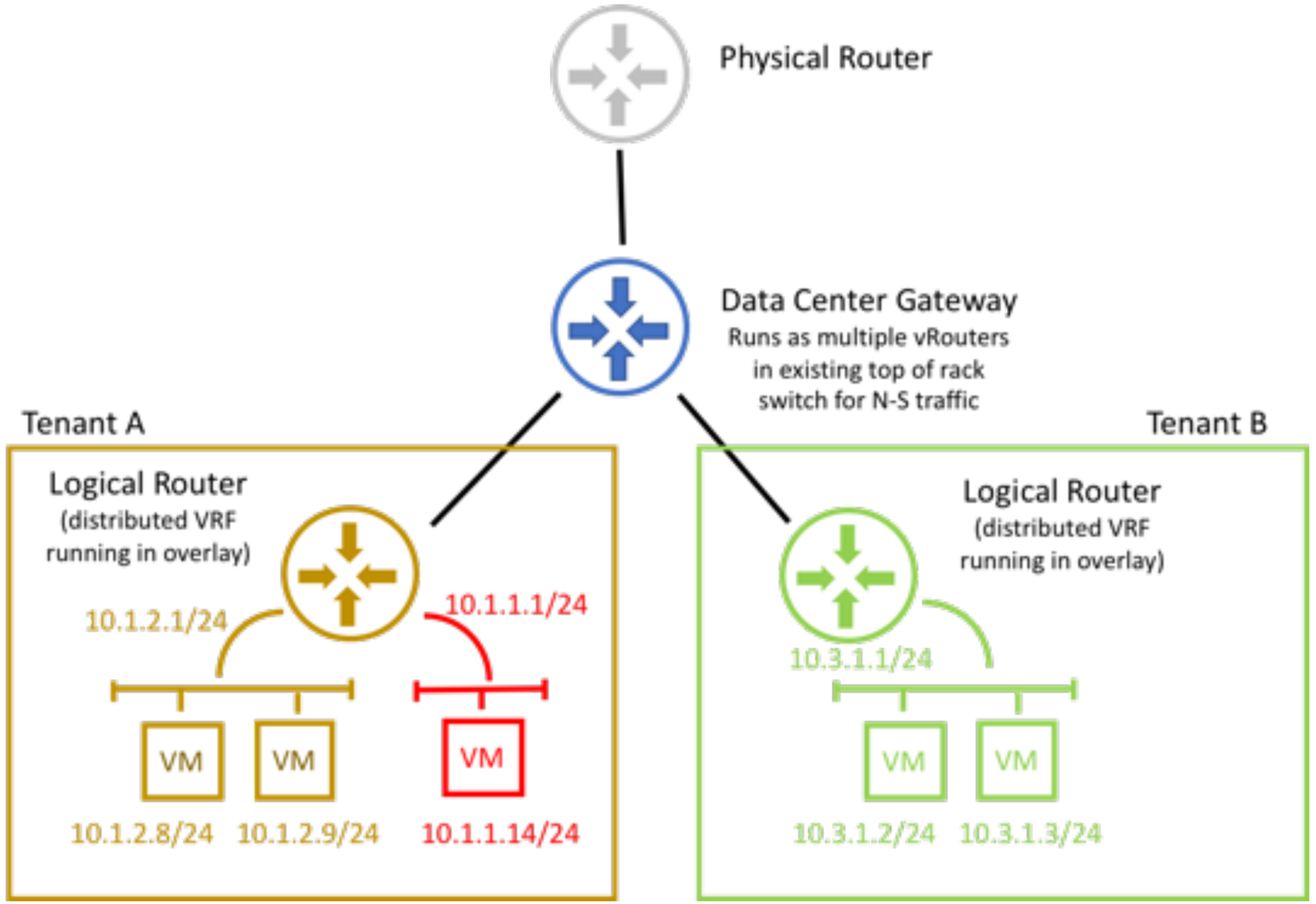
- Rely on physics
- Reduce fate sharing
- Improve encapsulation



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

Physical (and virtual) servers

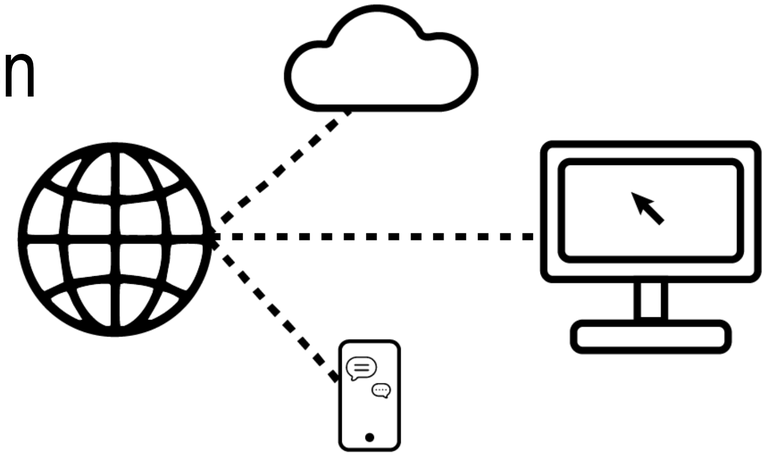
- Rely on physics
- Reduce fate sharing
- Improve encapsulation

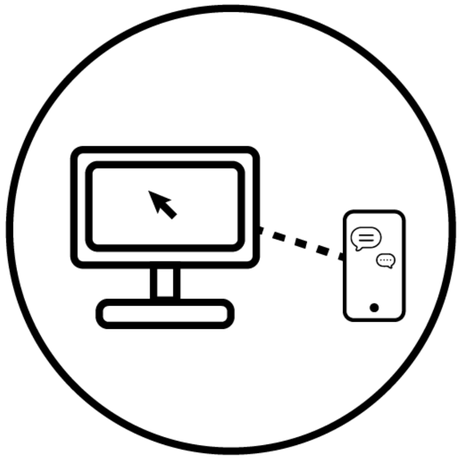


https://www.pluribusnetworks.com/blog/what-is-network-segmentation/

Physical (and virtual) servers

- Rely on physics
- Reduce fate sharing
- Improve encapsulation





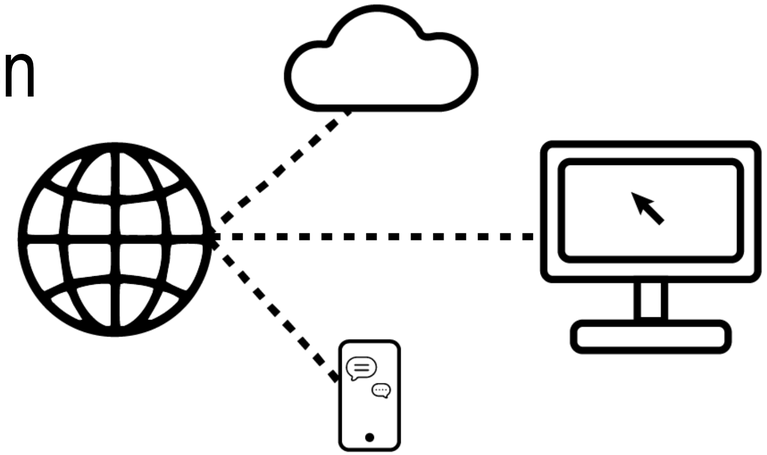
Air-gapped Network

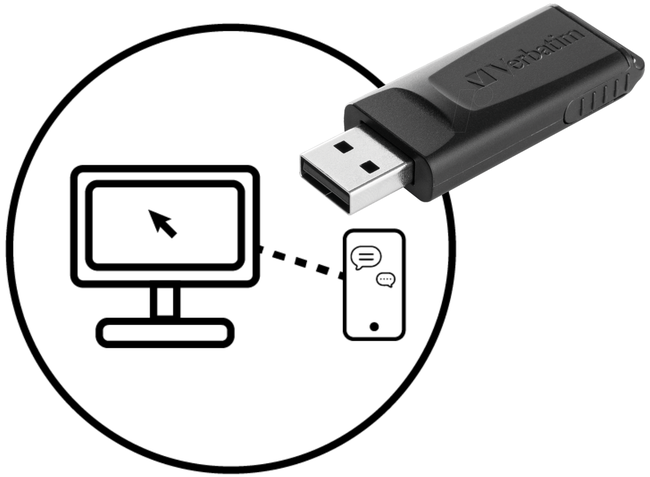
Devices included in the air-gapped network are physically isolated and can communicate with each other, but cannot communicate with any other network outside of the air-gap.

https://www.belden.com/hs-fs/hubfs/Arigap-Diagram-01.png

Physical (and virtual) servers

- Rely on physics
- Reduce fate sharing
- Improve encapsulation





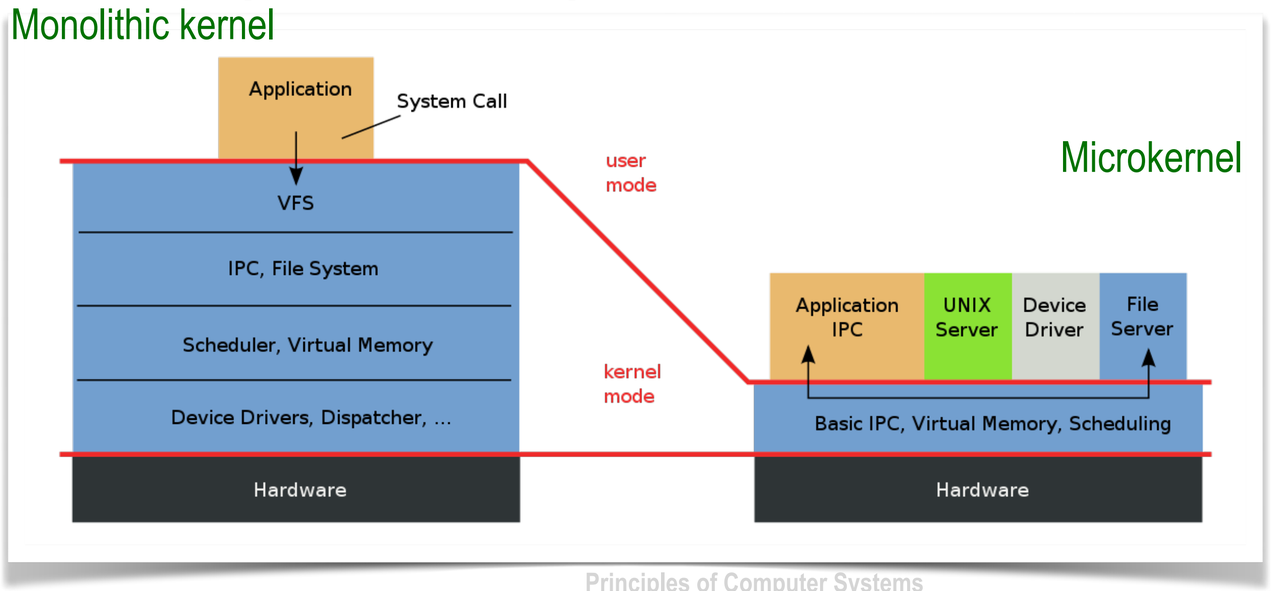
Air-gapped Network

Devices included in the air-gapped network are physically isolated and can communicate with each other, but cannot communicate with any other network outside of the air-gap.

https://www.belden.com/hs-fs/hubfs/Arigap-Diagram-01.png

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?

Exokernels

- An exercise in abstraction
  - *Exterminate all OS abstractions*
- Enable user space to safely implement new OS abstractions
- How is fate sharing? How is encapsulation?

Benefits of Client/Server

- Narrow channels for error propagation
  - *Isolation between "caller" and "callee"*

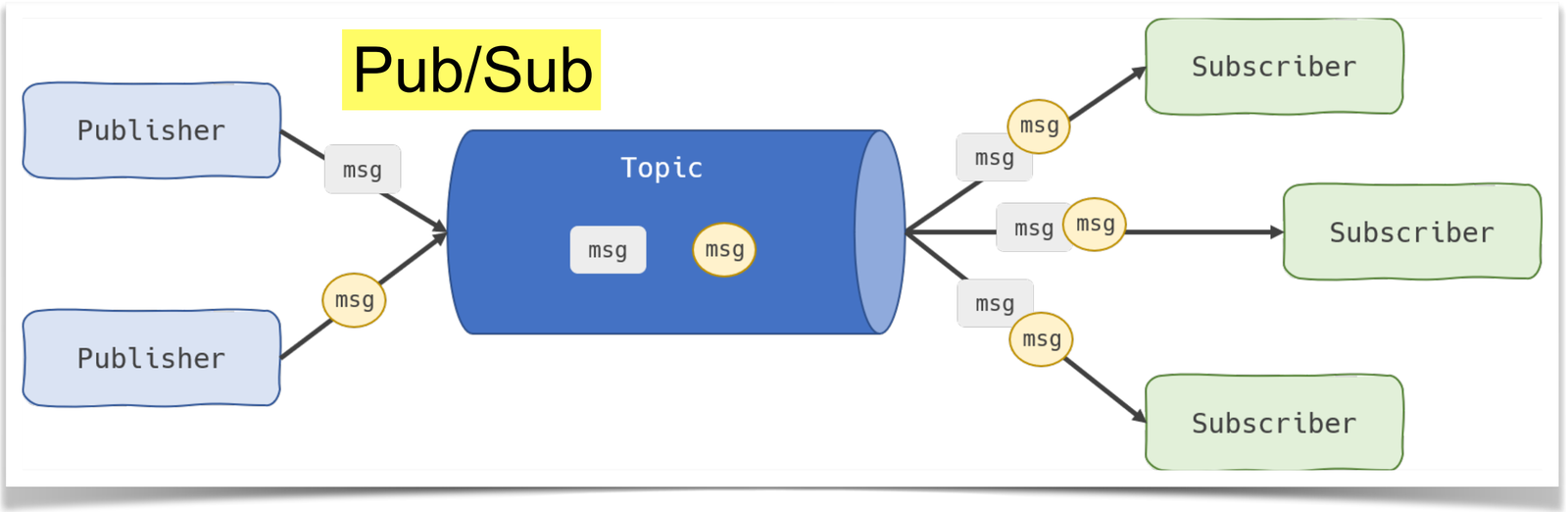
Benefits of Client/Server

- 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*
- 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*

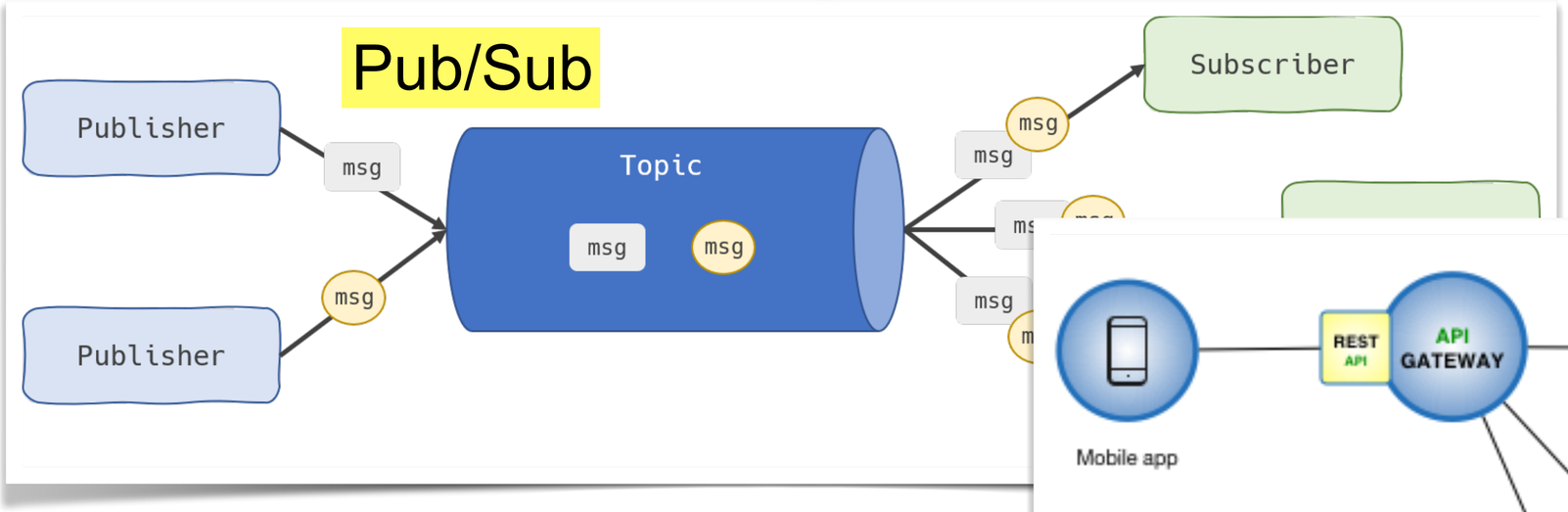
A couple of examples of client/server architectures

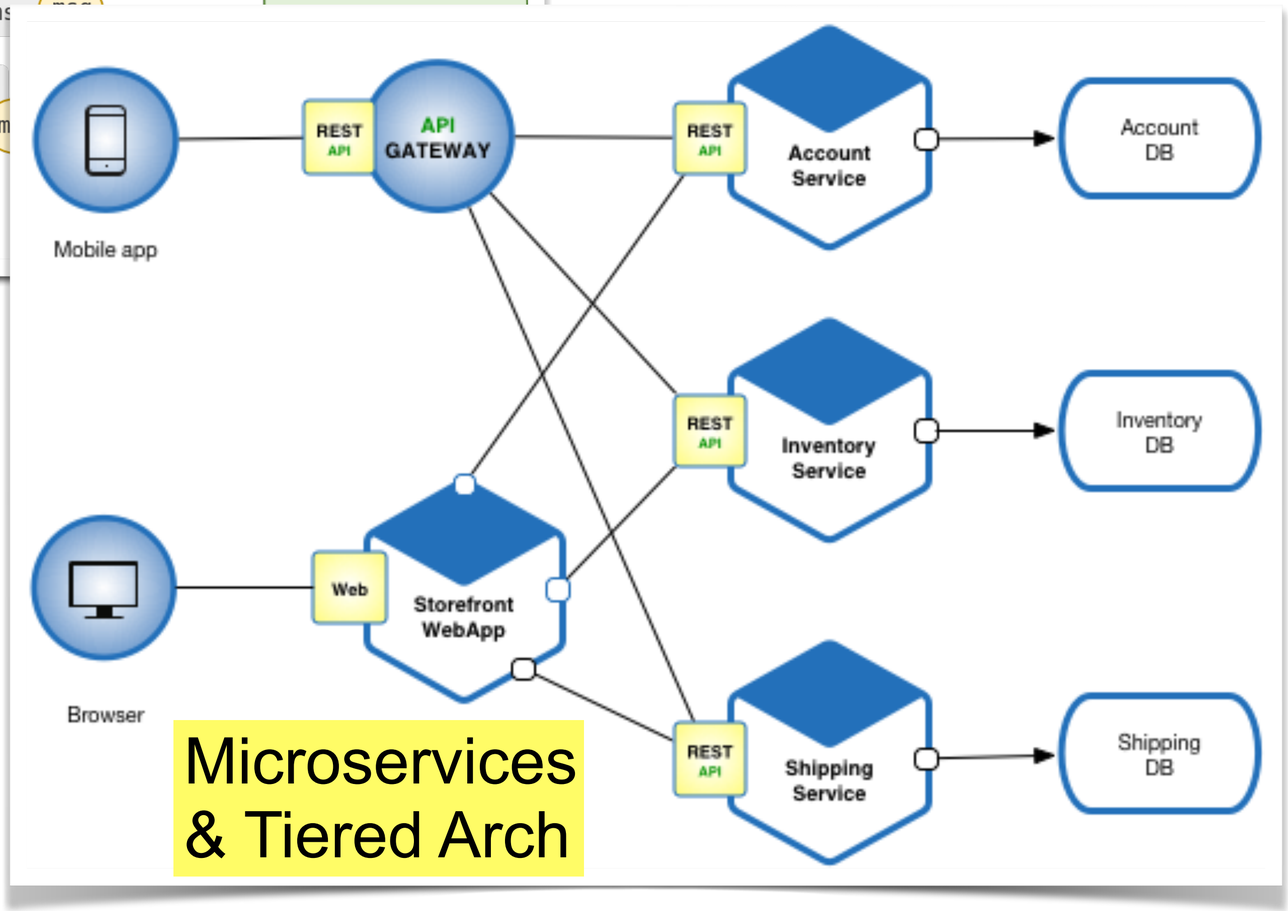


https://dashbird.io/knowledge-base/well-architected/pub-sub-messaging/

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

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

Memory safety

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

Message-based communication

Remote Procedure Calls (RCP)

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

Mechanics of RPC

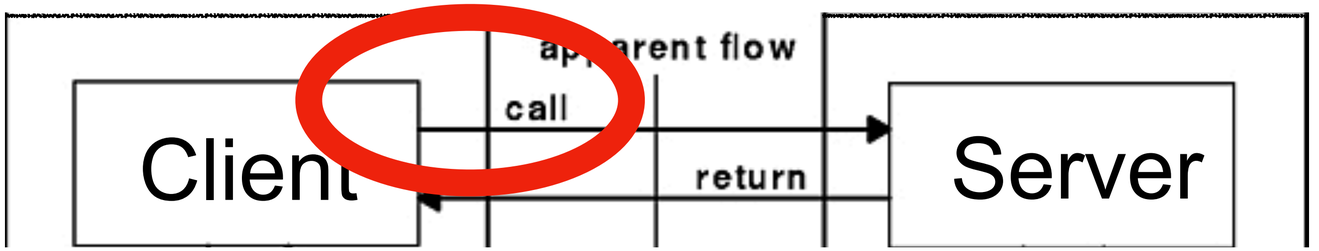


Image courtesy of https://www.ibm.com/support/knowledgecenter

Mechanics of RPC

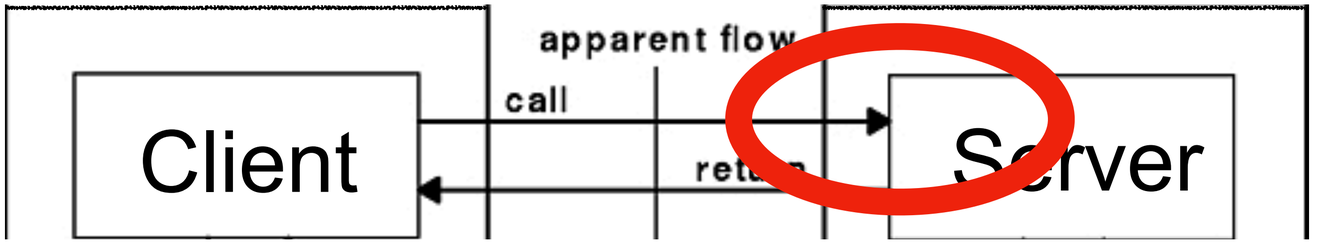


Image courtesy of https://www.ibm.com/support/knowledgecenter

Mechanics of RPC

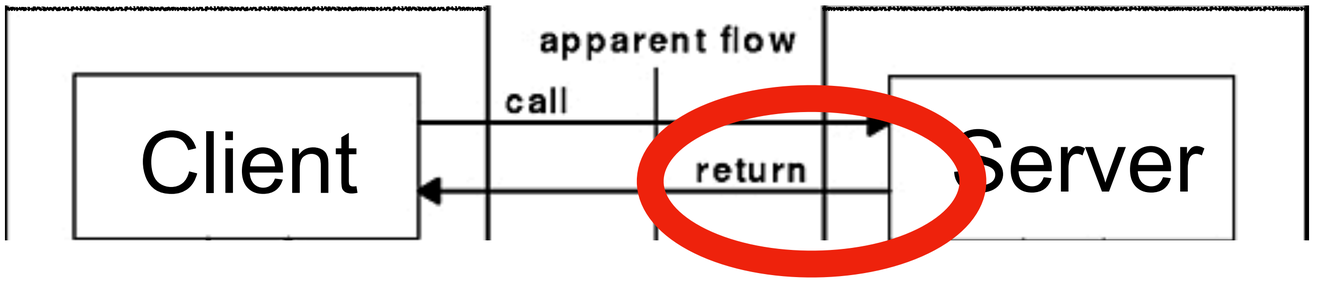


Image courtesy of https://www.ibm.com/support/knowledgecenter

Mechanics of RPC

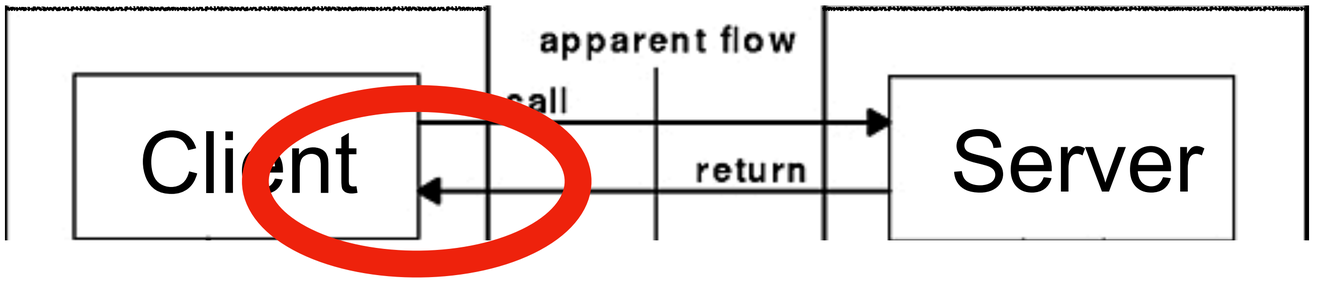


Image courtesy of https://www.ibm.com/support/knowledgecenter

Mechanics of RPC

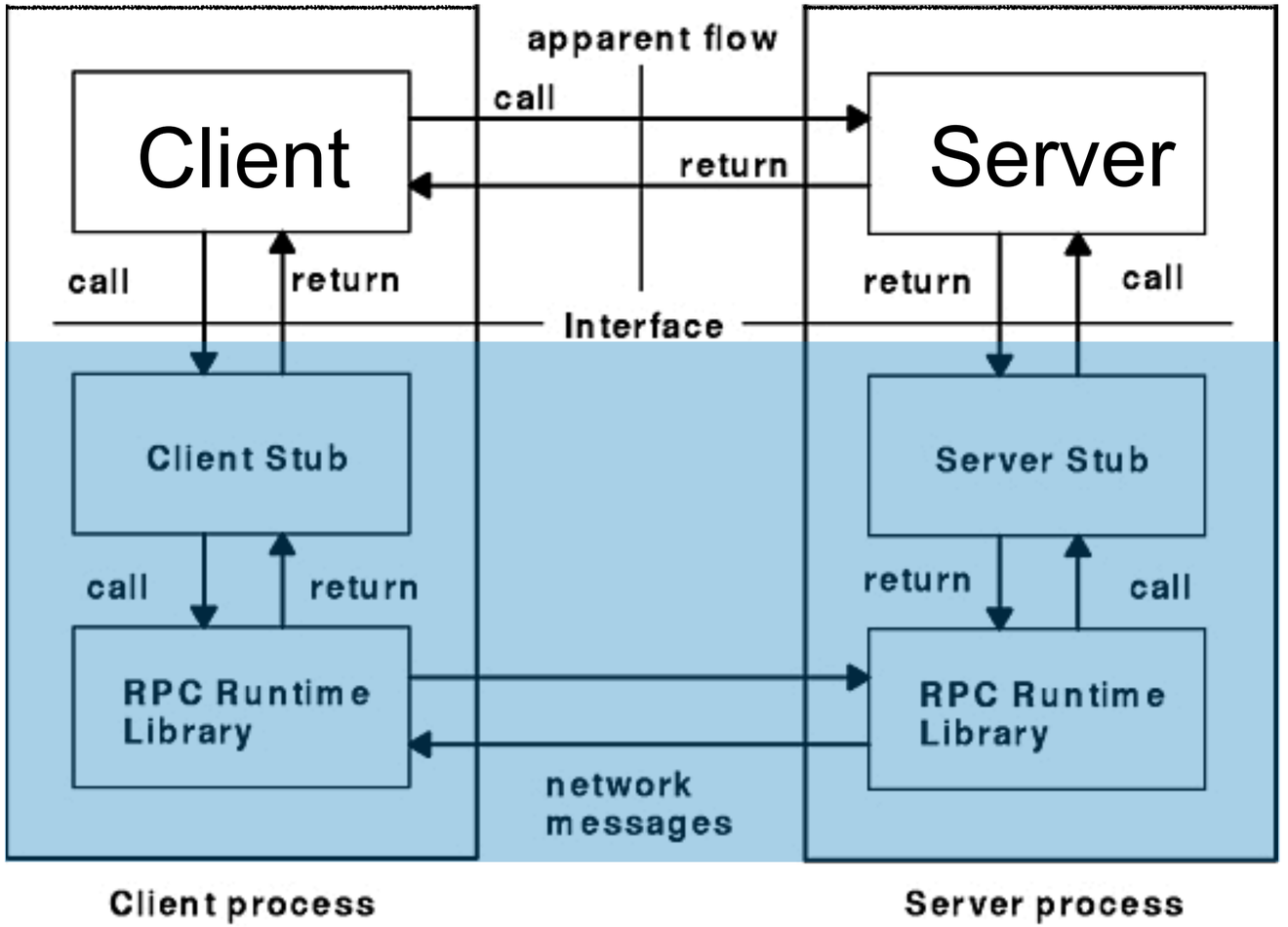
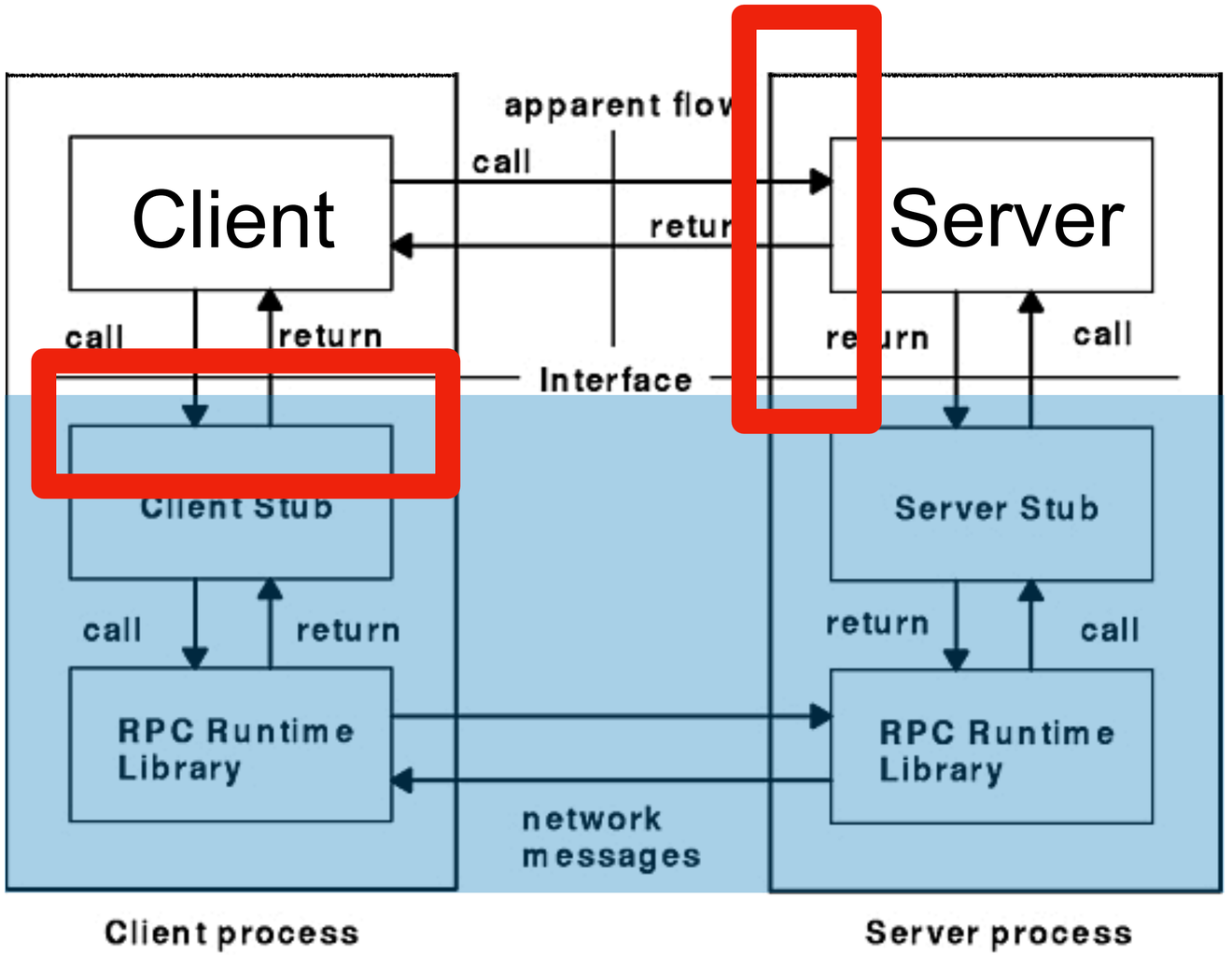
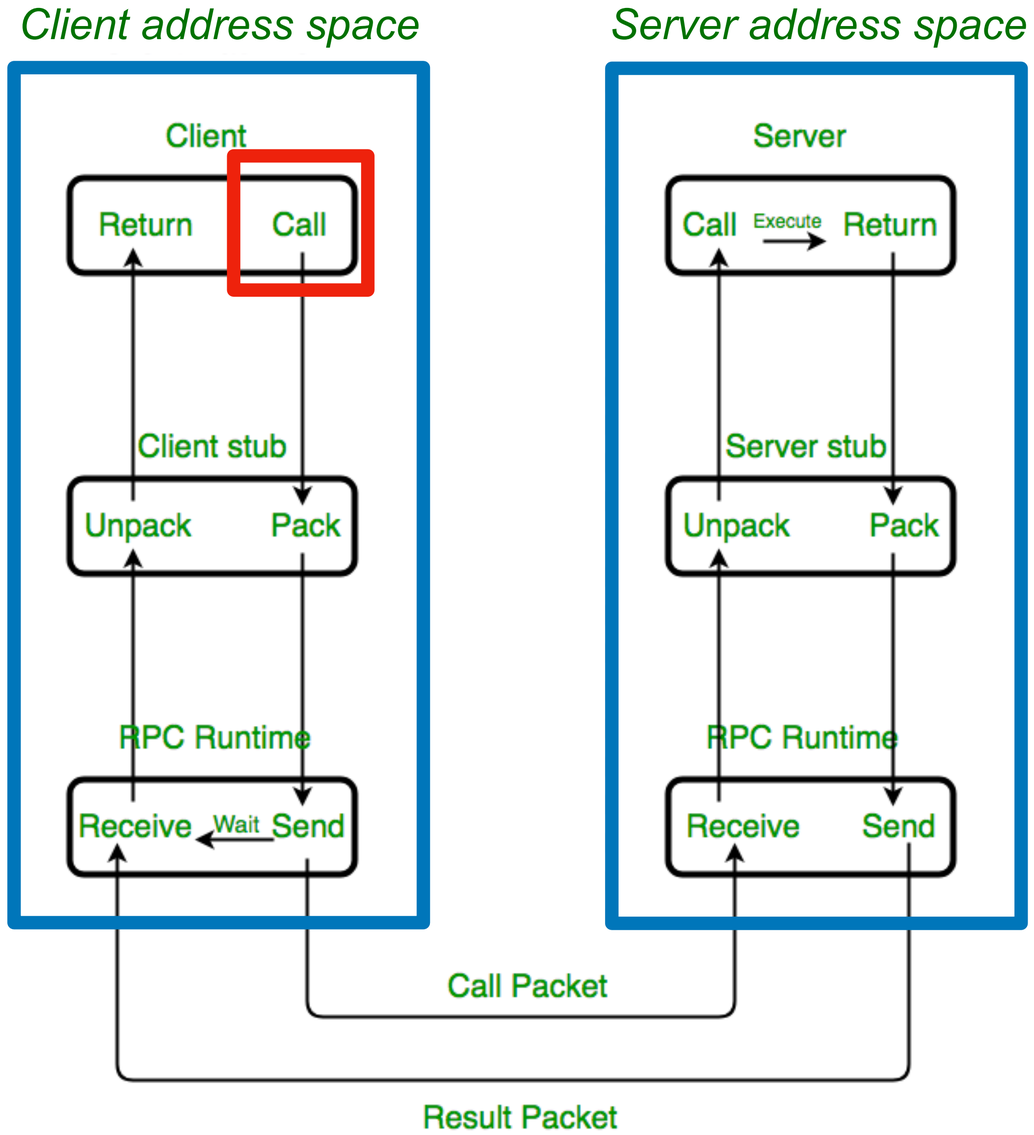


Image courtesy of https://www.ibm.com/support/knowledgecenter

Mechanics of RPC



Mechanics of RPC



Local procedure call

Parameters -> message

Message -> packet

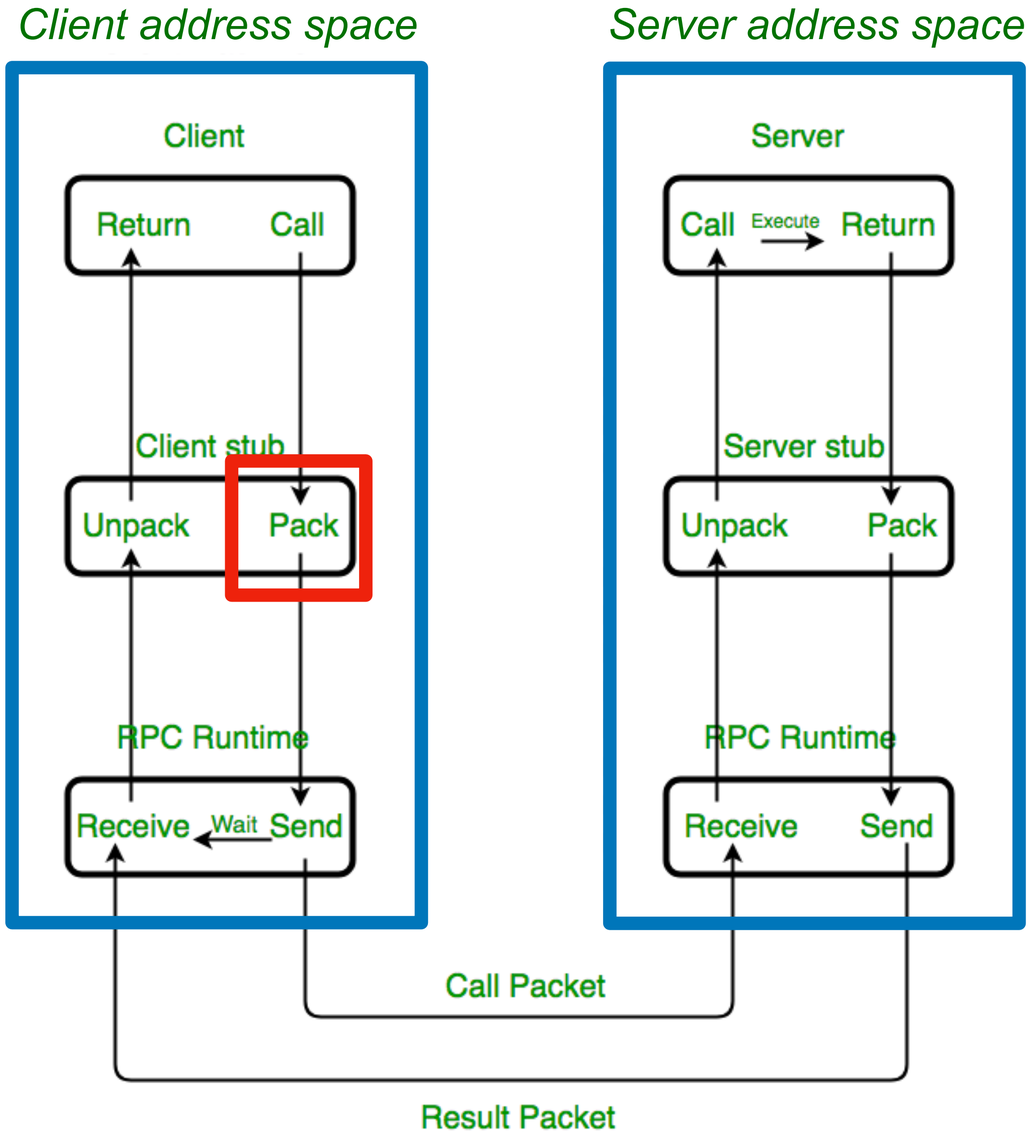
Local procedure invocation

Message -> parameters

Packet -> message

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

Mechanics of RPC



Local procedure call

Parameters -> message

Message -> packet

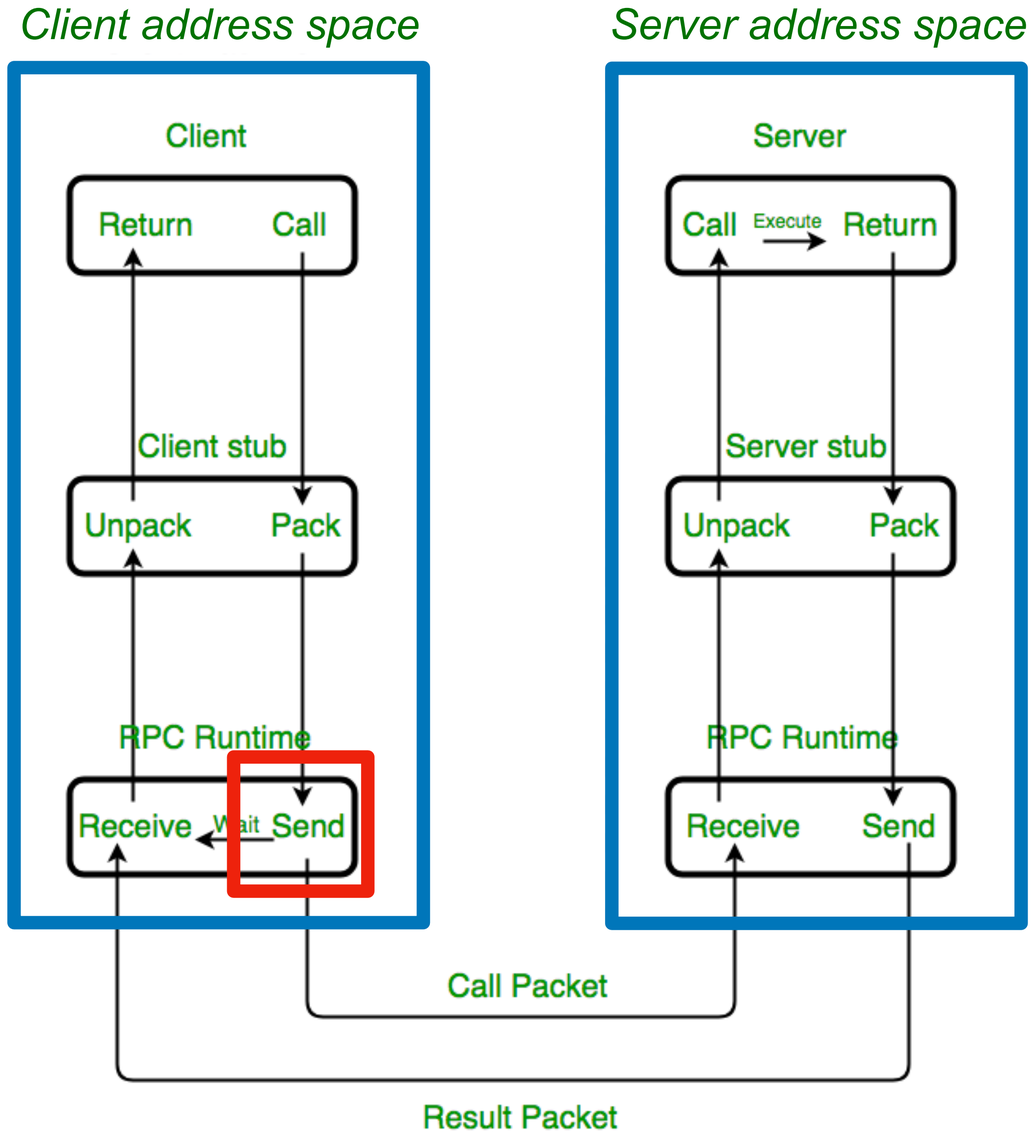
Local procedure invocation

Message -> parameters

Packet -> message

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

Mechanics of RPC



Local procedure call

Parameters -> message

Message -> packet

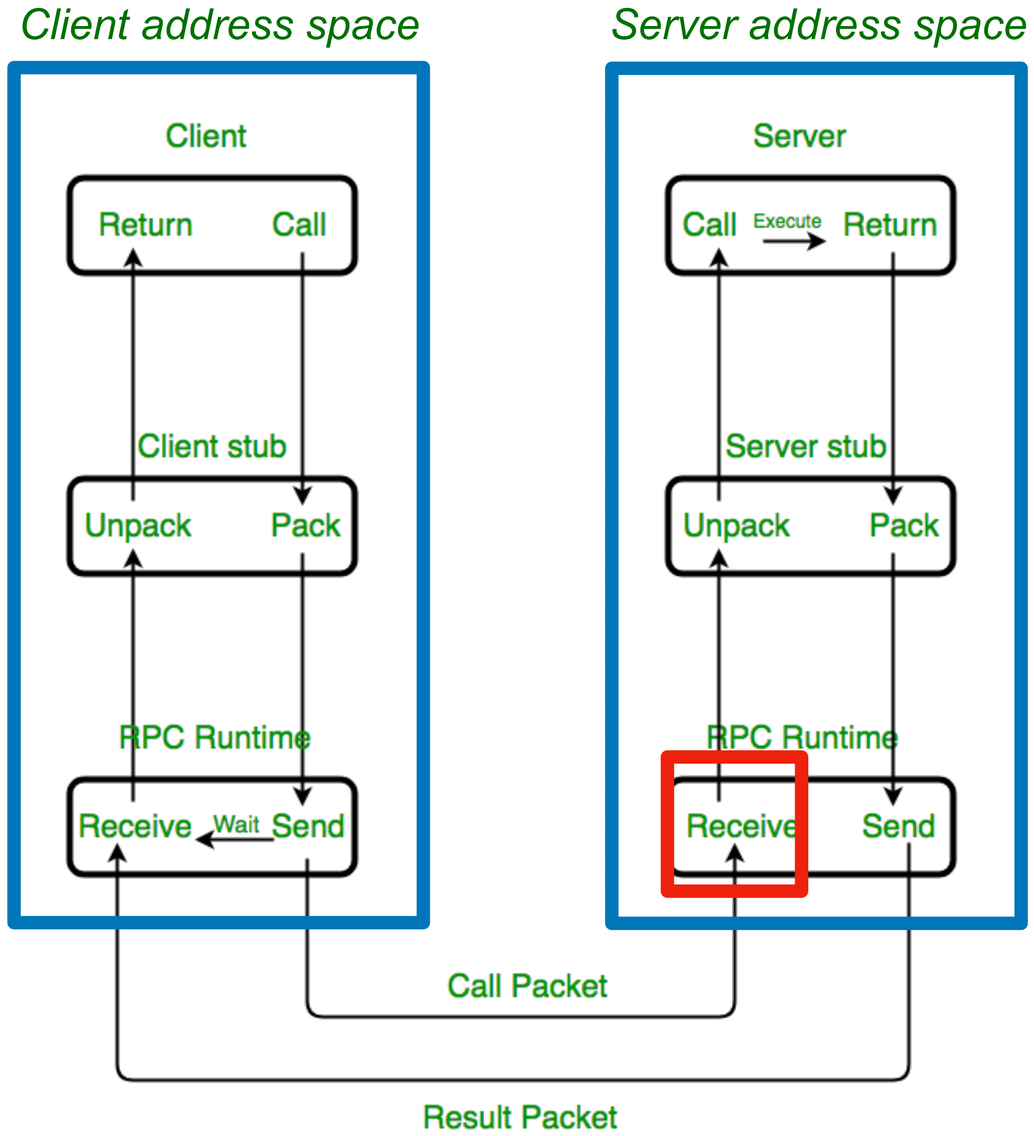
Local procedure invocation

Message -> parameters

Packet -> message

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

Mechanics of RPC



Local procedure call

Parameters -> message

Message -> packet

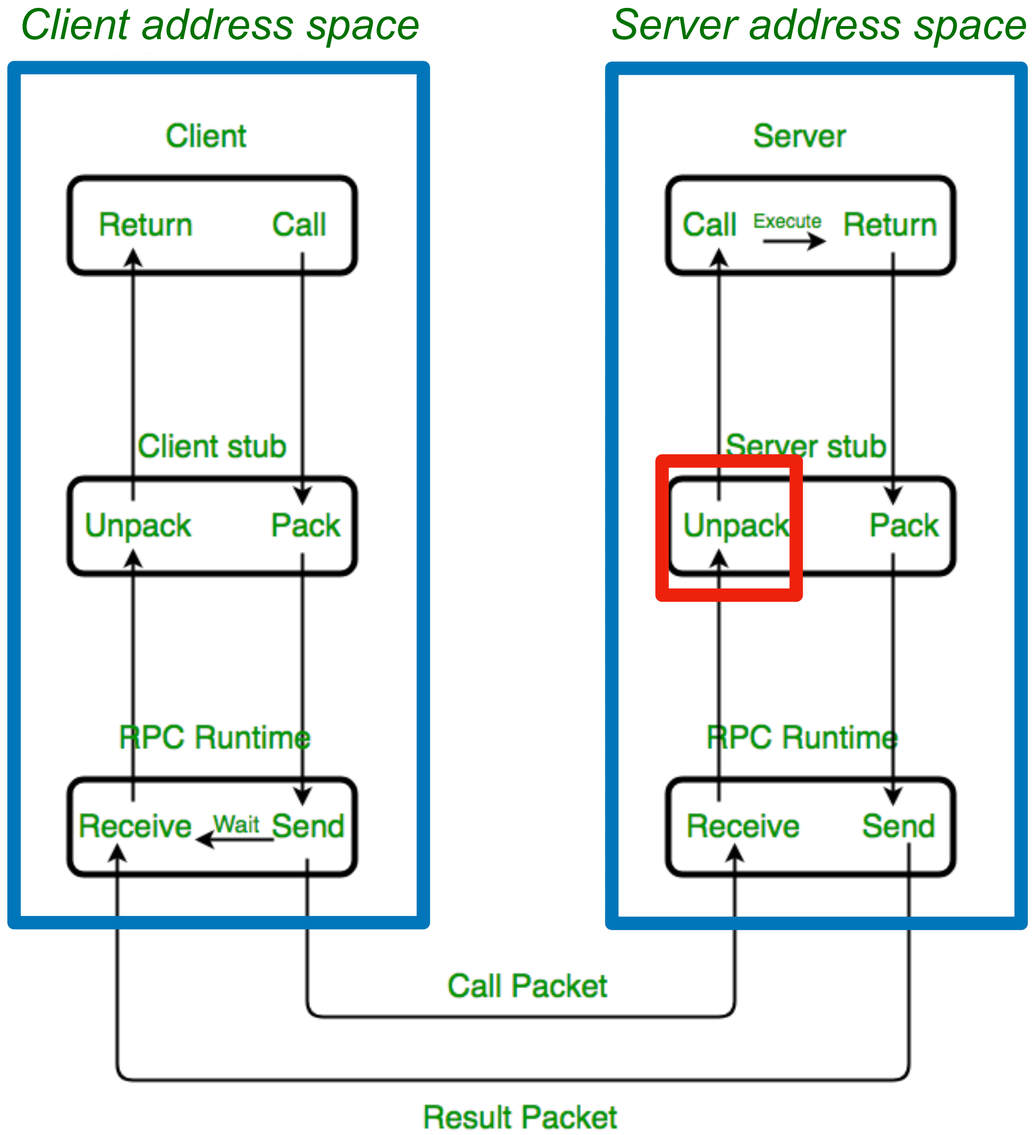
Local procedure invocation

Message -> parameters

Packet -> message

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

Mechanics of RPC



Local procedure call

Parameters -> message

Message -> packet

Local procedure invocation

Message -> parameters

Packet -> message

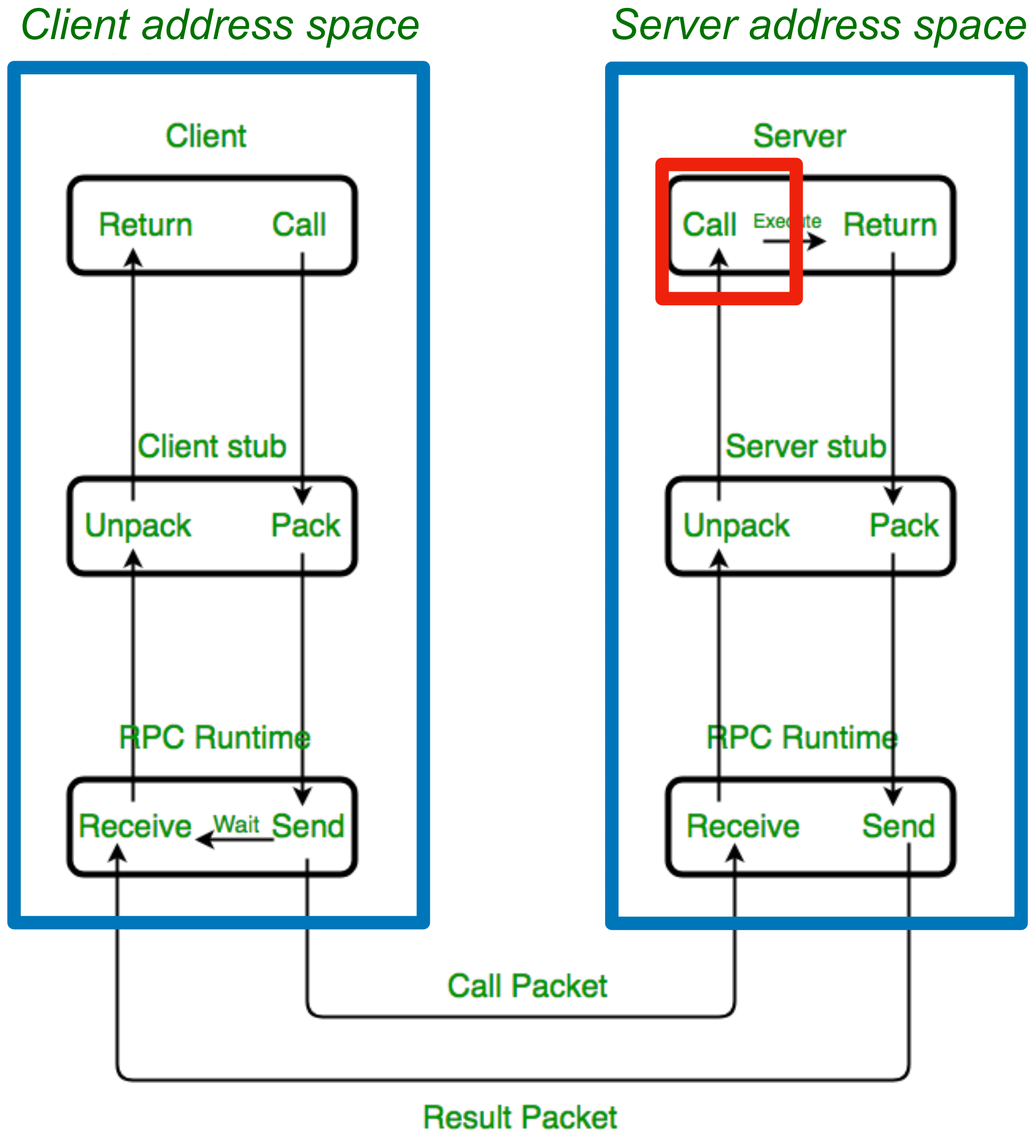
Image courtesy of https://www.geeksforgeeks.org/remote-procedure-call-rpc-in-operating-system/

Mechanics of RPC

Local procedure call

Parameters -> message

Message -> packet



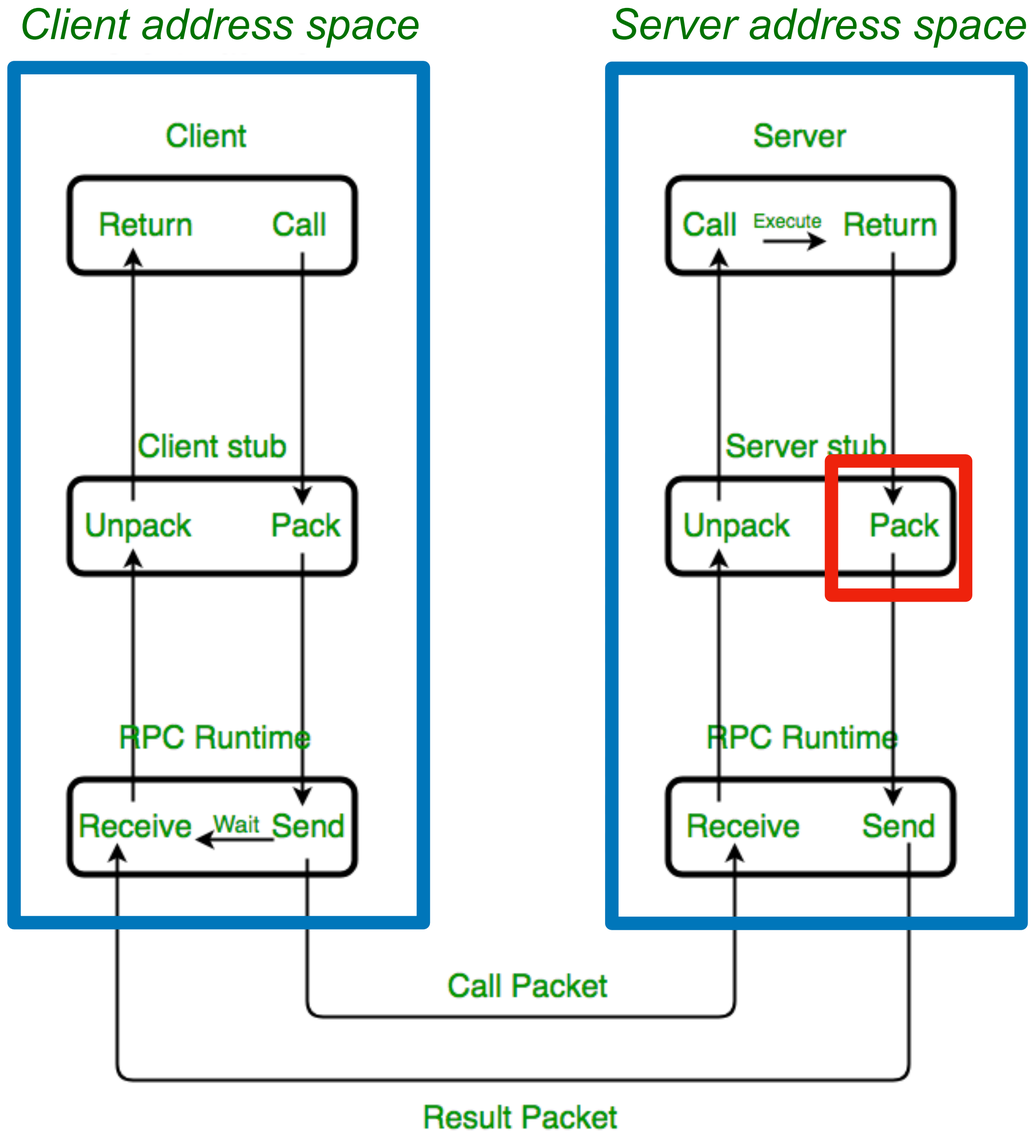
Local procedure invocation

Message -> parameters

Packet -> message

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

Mechanics of RPC



Local procedure call

Parameters -> message

Message -> packet

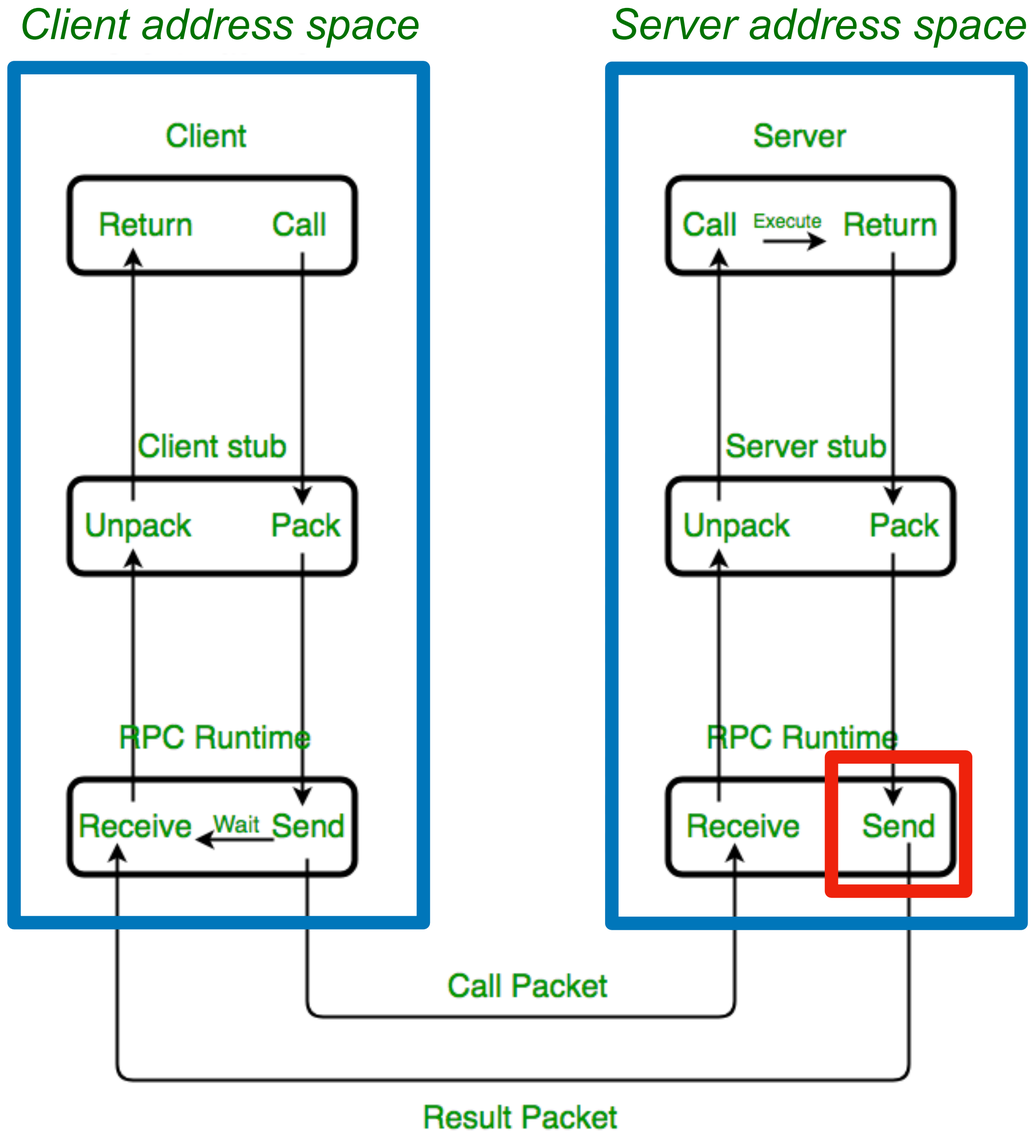
Local procedure invocation

Message -> parameters

Packet -> message

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

Mechanics of RPC



Local procedure call

Parameters -> message

Message -> packet

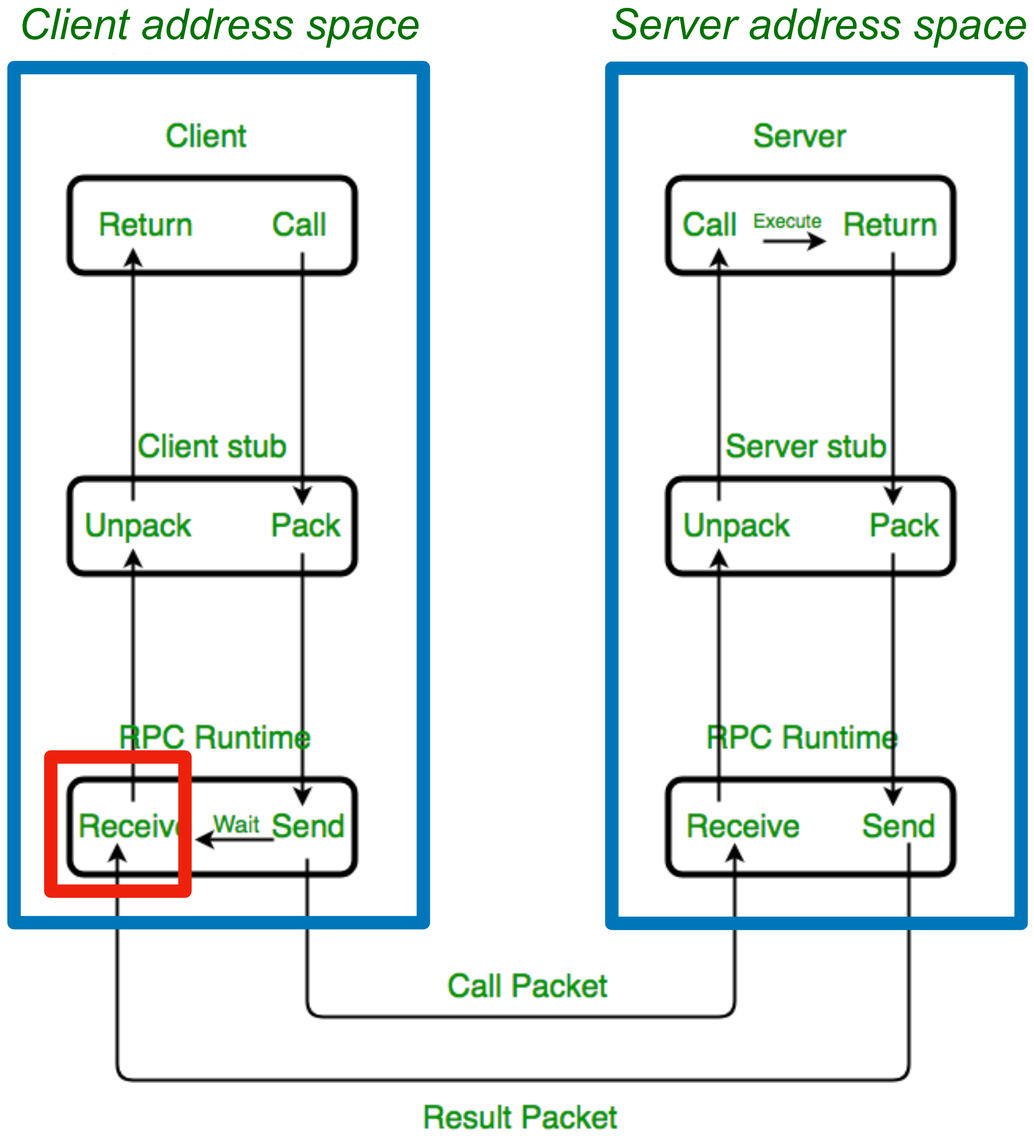
Local procedure invocation

Message -> parameters

Packet -> message

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

Mechanics of RPC



Local procedure call

Parameters -> message

Message -> packet

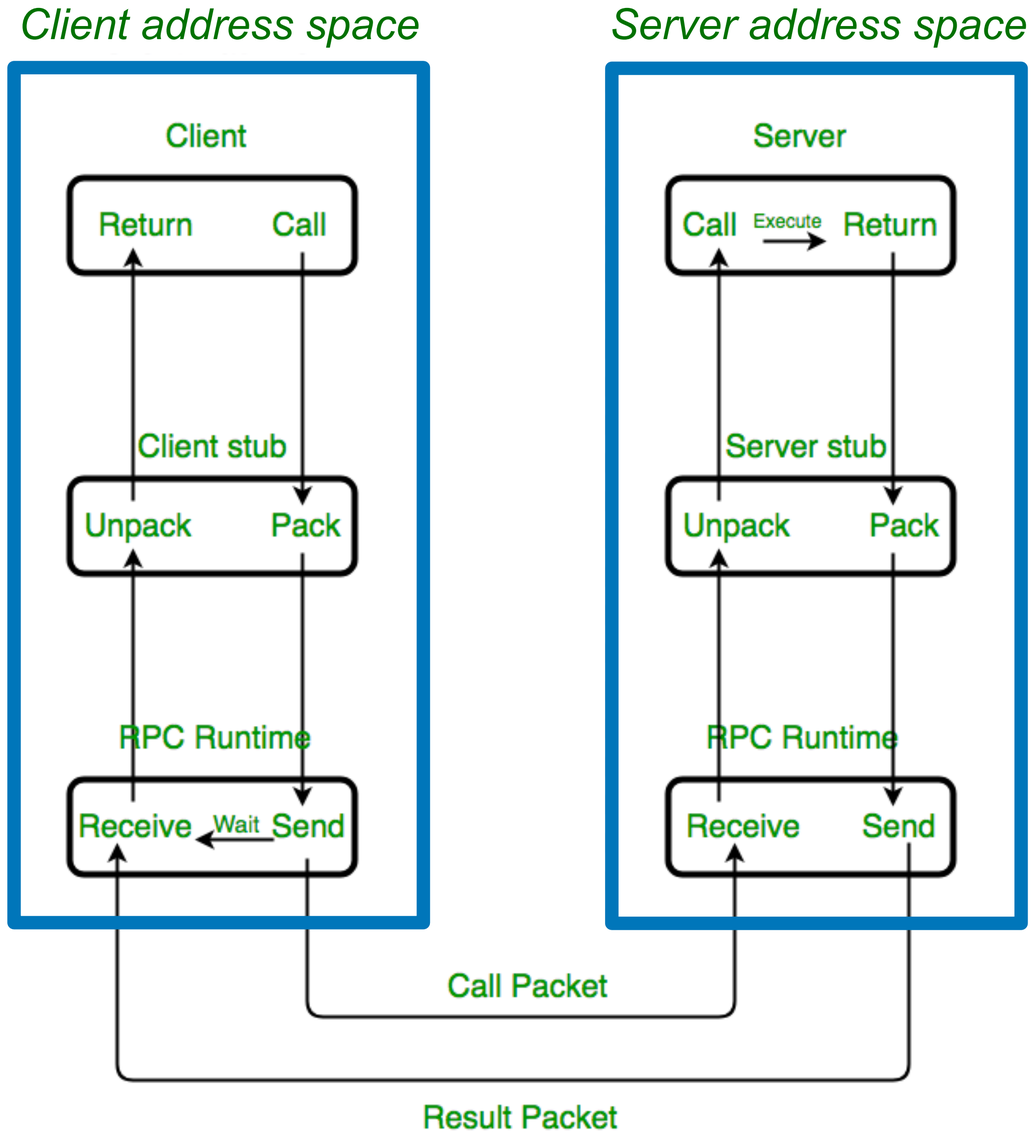
Local procedure invocation

Message -> parameters

Packet -> message

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

Mechanics of RPC



Local procedure call

Parameters -> message

Message -> packet

Local procedure invocation

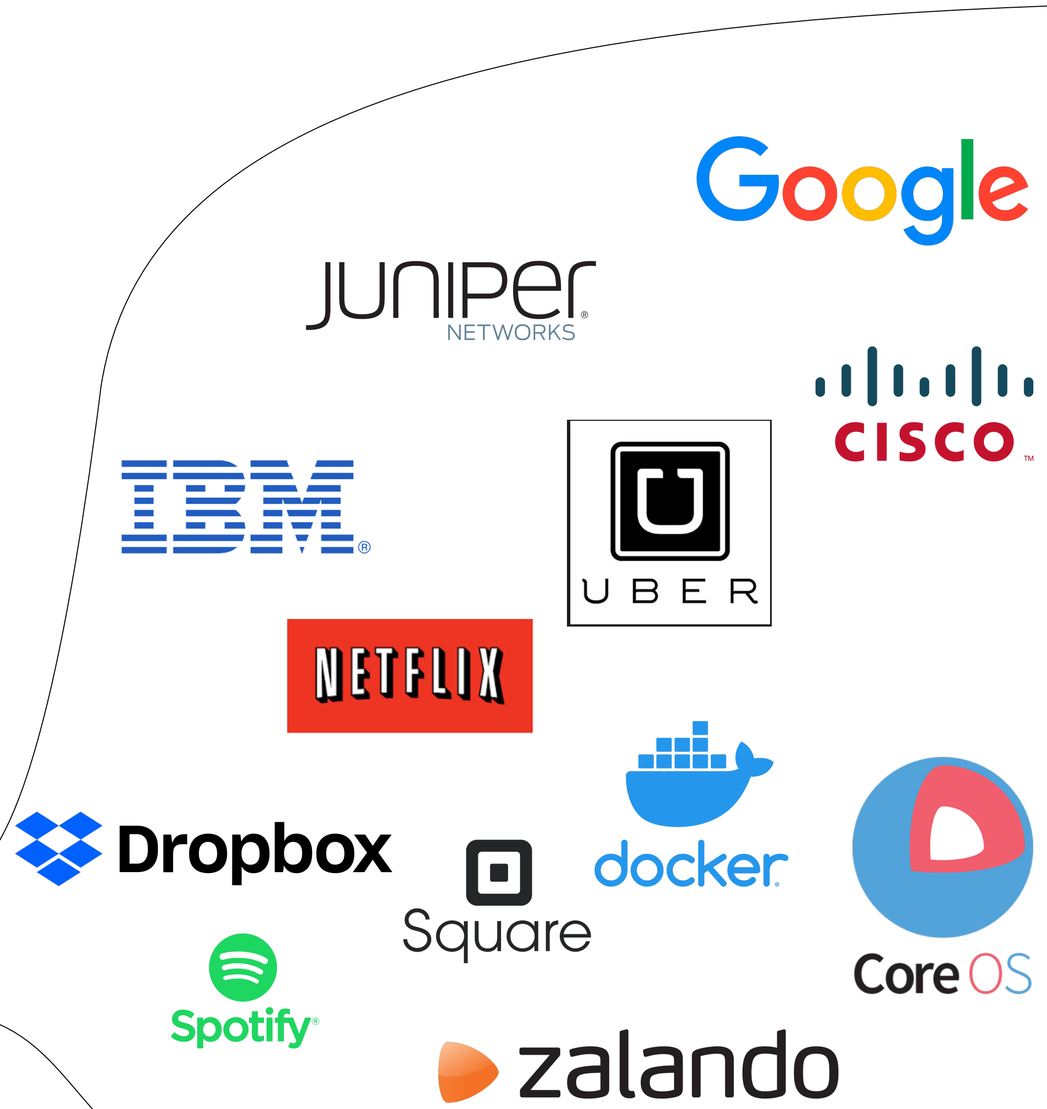
Message -> parameters

Packet -> message

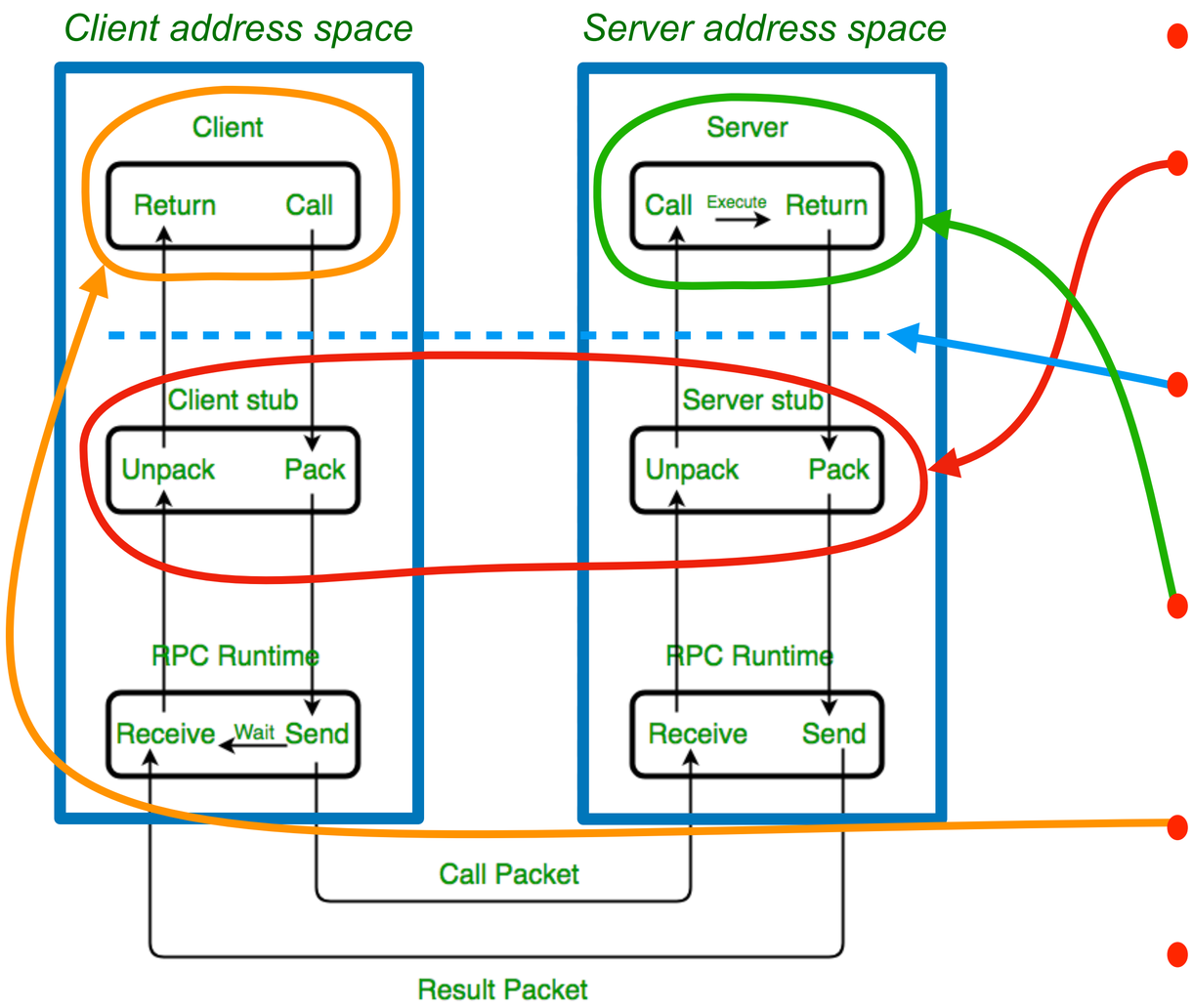
Image courtesy of https://www.geeksforgeeks.org/remote-procedure-call-rpc-in-operating-system/

Examples of RPC systems

- NFS
- Java RMI
- Package `rpc` in Go
- Google Web Toolkit
- SOAP (successor to XML-RPC)
- Apache Thrift
- gRPC (uses Google Protocol Buffers IDL)



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

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?

No response from RPC = ?

- At-least-once semantics
- At-most-once semantics
- Exactly-once semantics

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)*

Recap

same address space

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

Memory safety

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

Message-based communication

separate address spaces