Modularity through Client/Server Organization

Prof. George Candea
School of Computer & Communication Sciences

Building blocks	
	~~Modules and interfaces~~
	~~Names~~
	~~Layers~~
	Client/server

Lecture objectives

- Understand effective techniques for modularizing systems
- Think (more) deeply about the trade-offs involved in modularization
  - *in order to resolve these trade-offs in an informed manner*
- Identify further examples of modularization
  - *understand how modularization differs from abstraction*
  - *understand the role that naming plays in modularization*

Outline

- Brief recap of modularization
- Client/server organization
- Remote procedure calls (RPC)

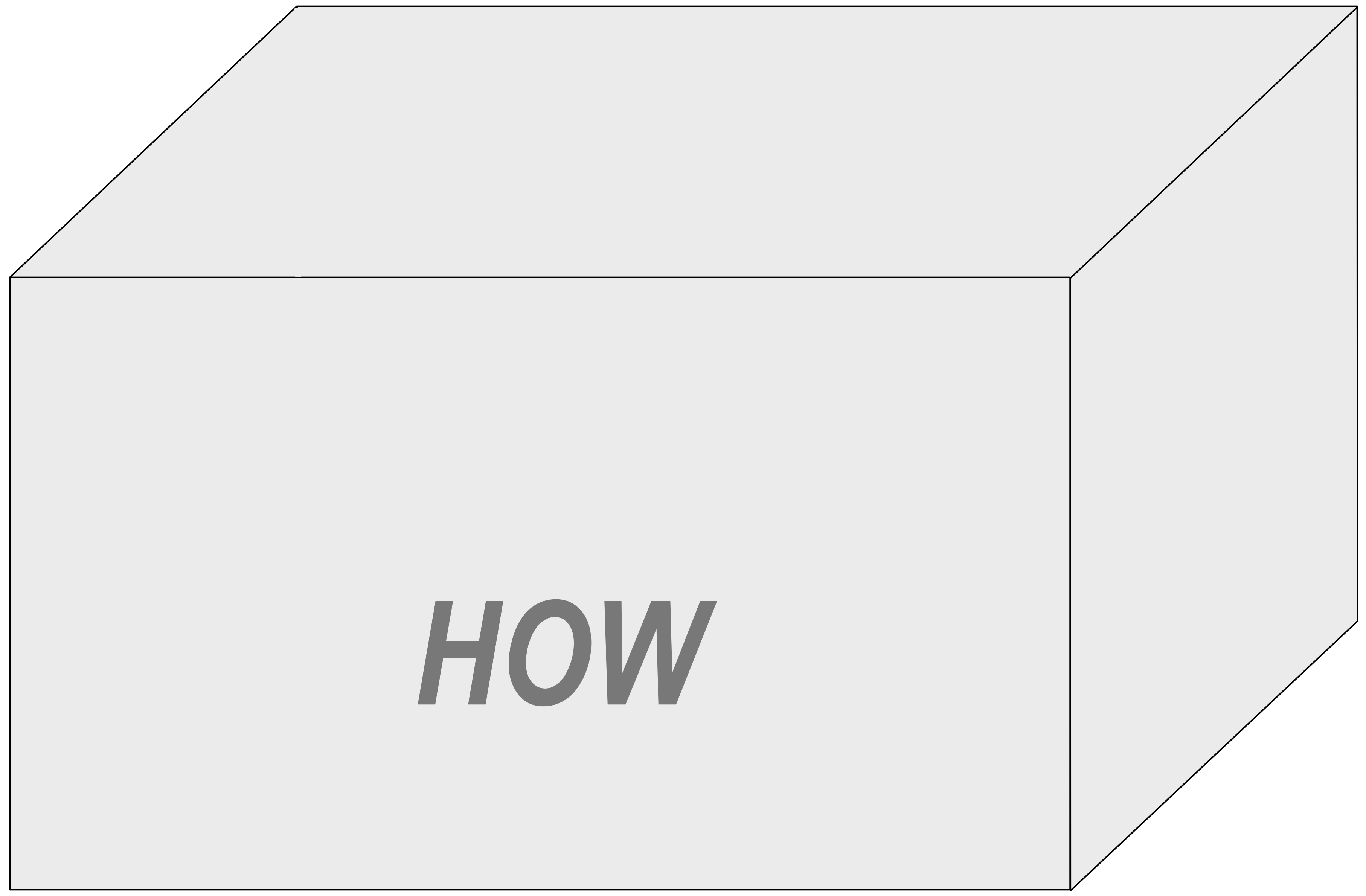
Recap of Modularization

Modularity

Modularity

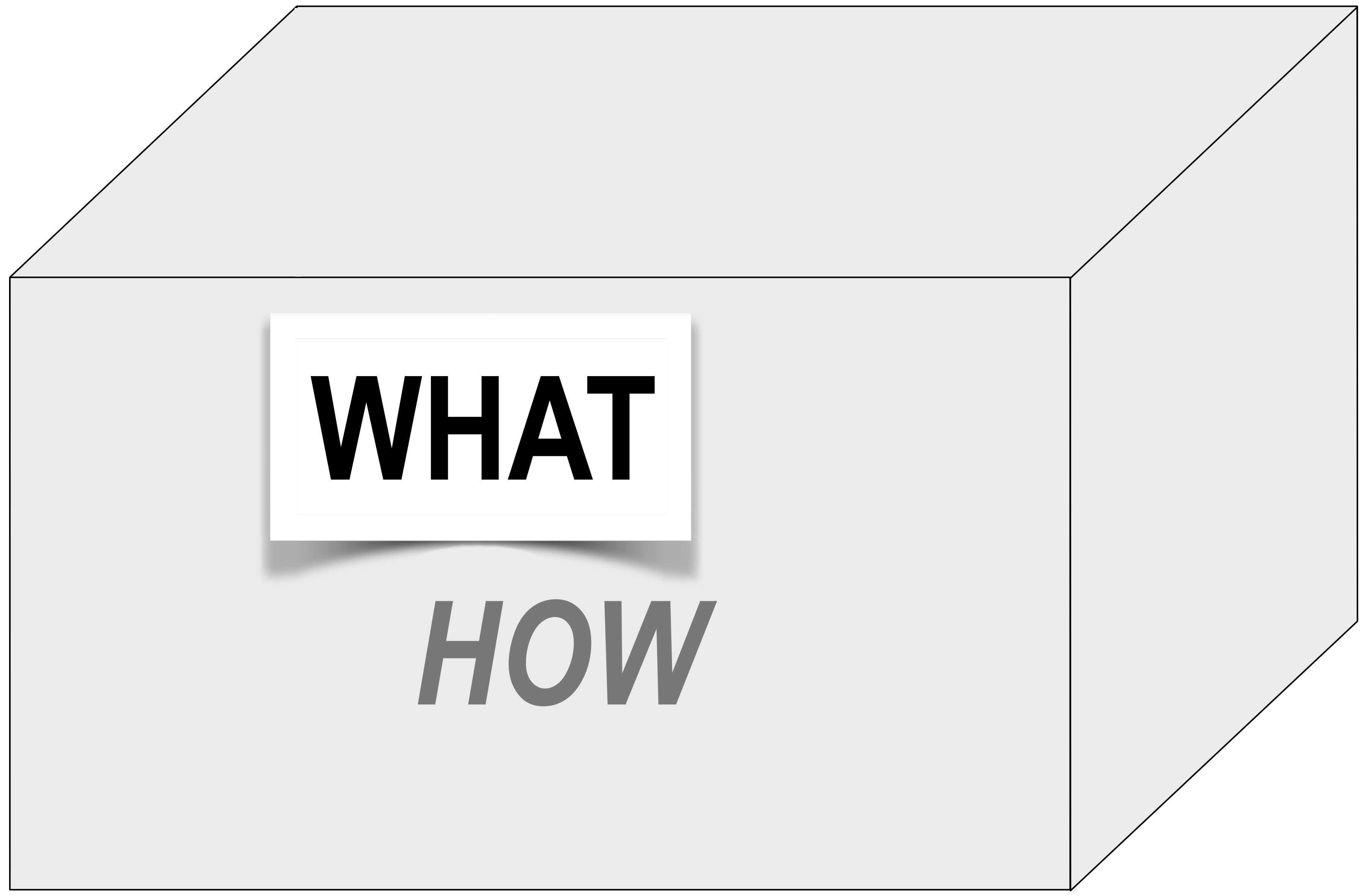
Abstraction

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



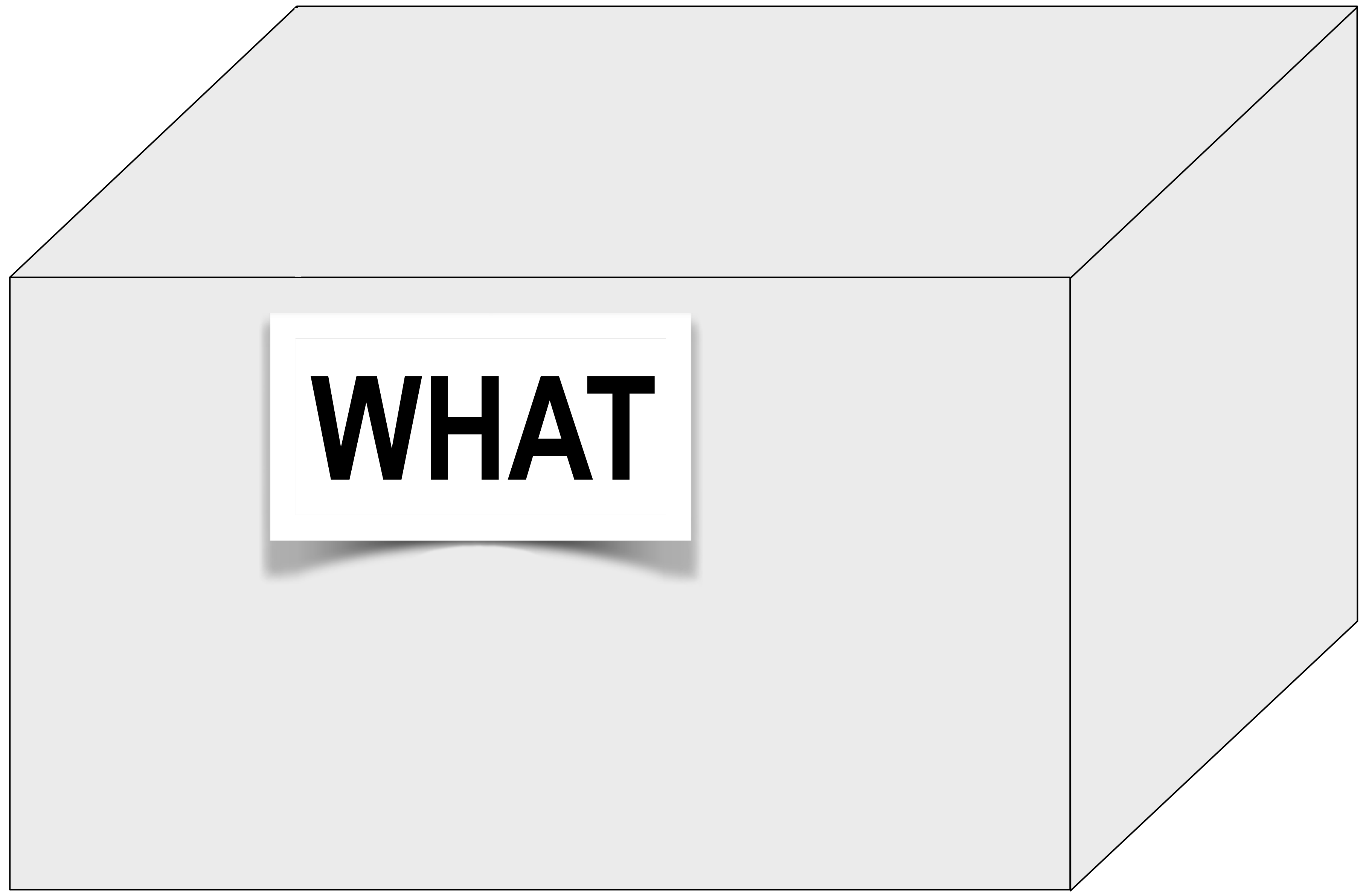
Abstraction

- Specifies "what" a component/subsystem does
- Together with modularity,
  it separates "what" from "how"
  => abstraction



Abstraction

- Specifies "what" a component/subsystem does
- Together with modularity,
  it separates "what" from "how"
  => abstraction

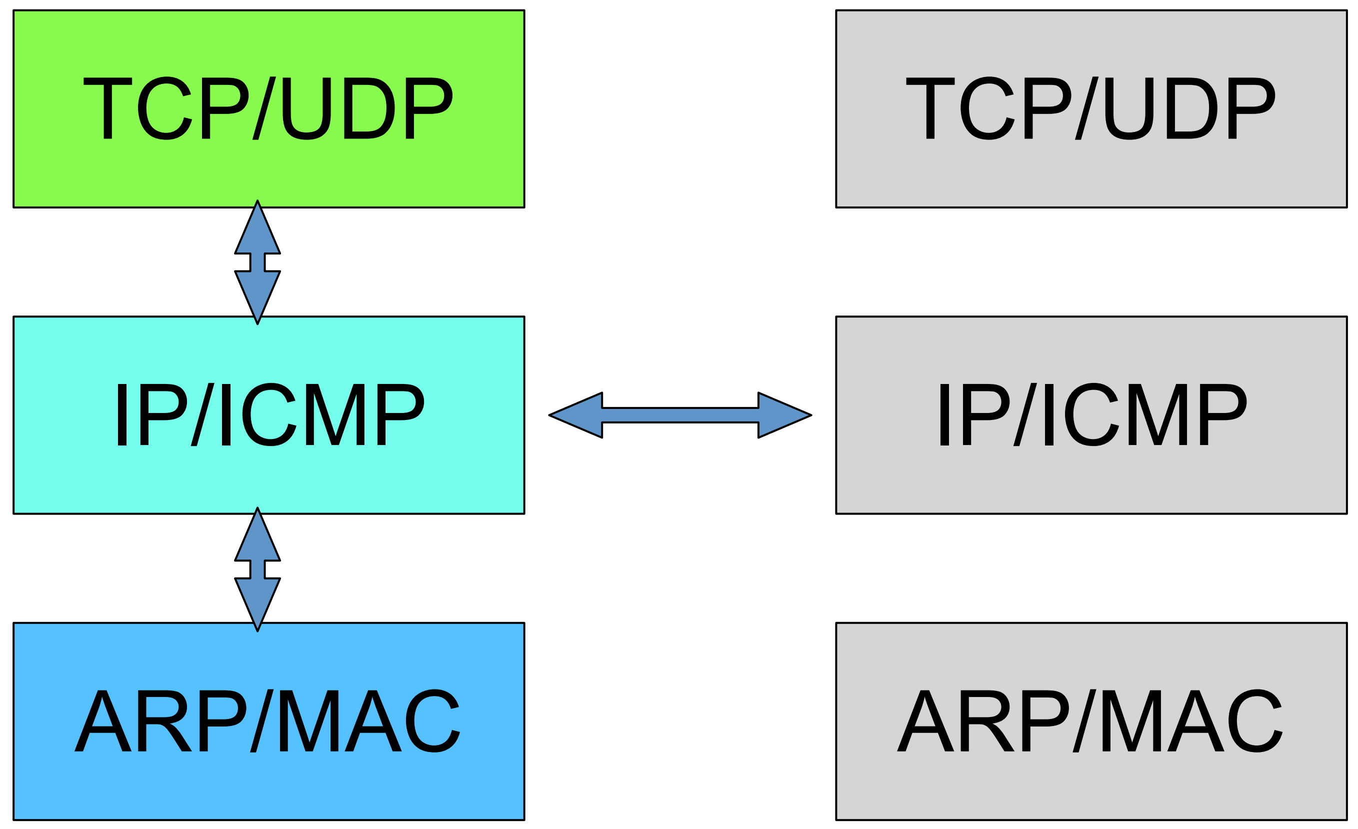


Names

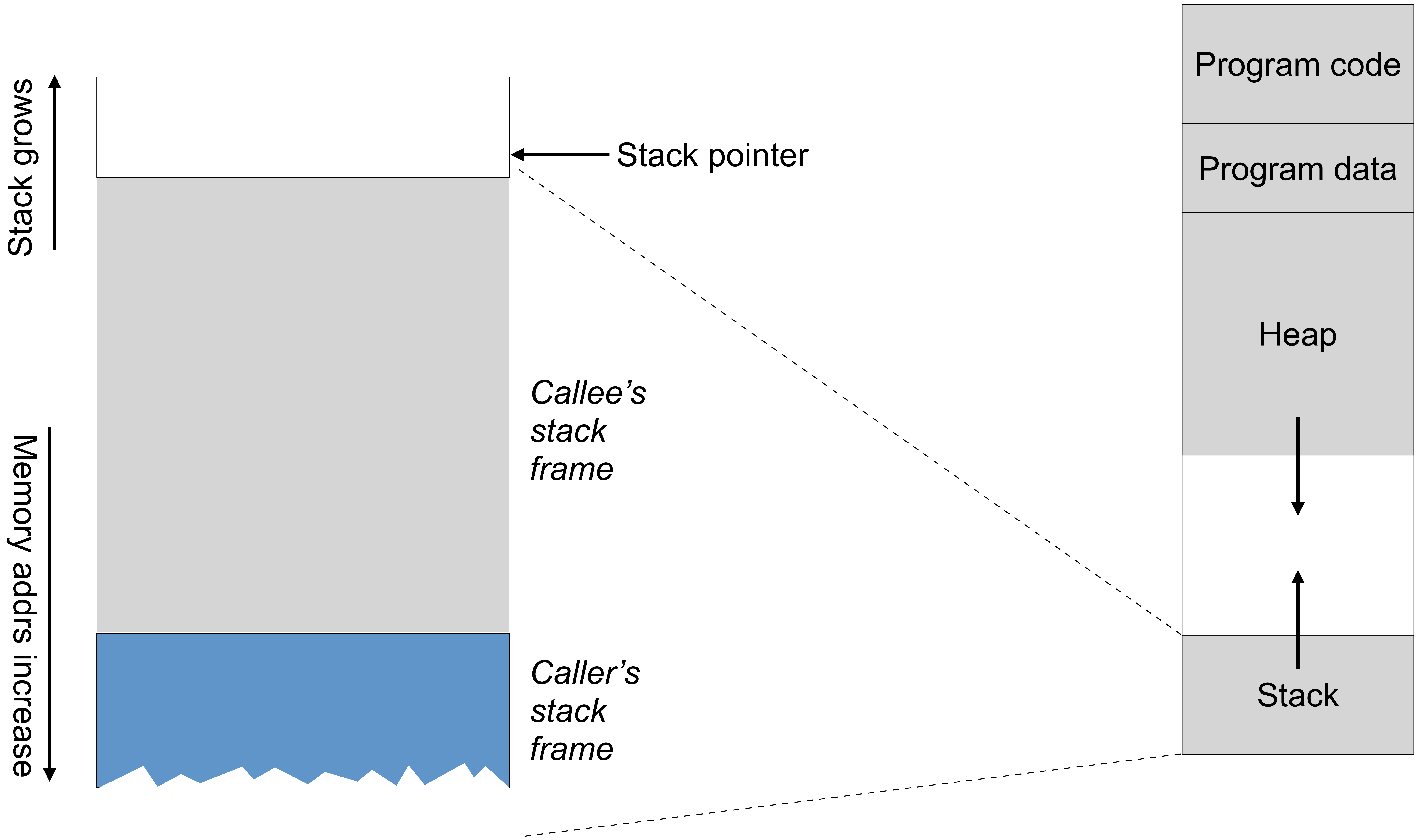
- Scope
  - *Private: unique within a context (e.g., a private IP address)*
  - *Global: unique across contexts (e.g., a global IP address)*
- Structure
  - *Hierarchical: name relationship implies object relationship (e.g., two IP addresses sharing the same prefix)*
  - *Flat: name relationship implies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

Layers

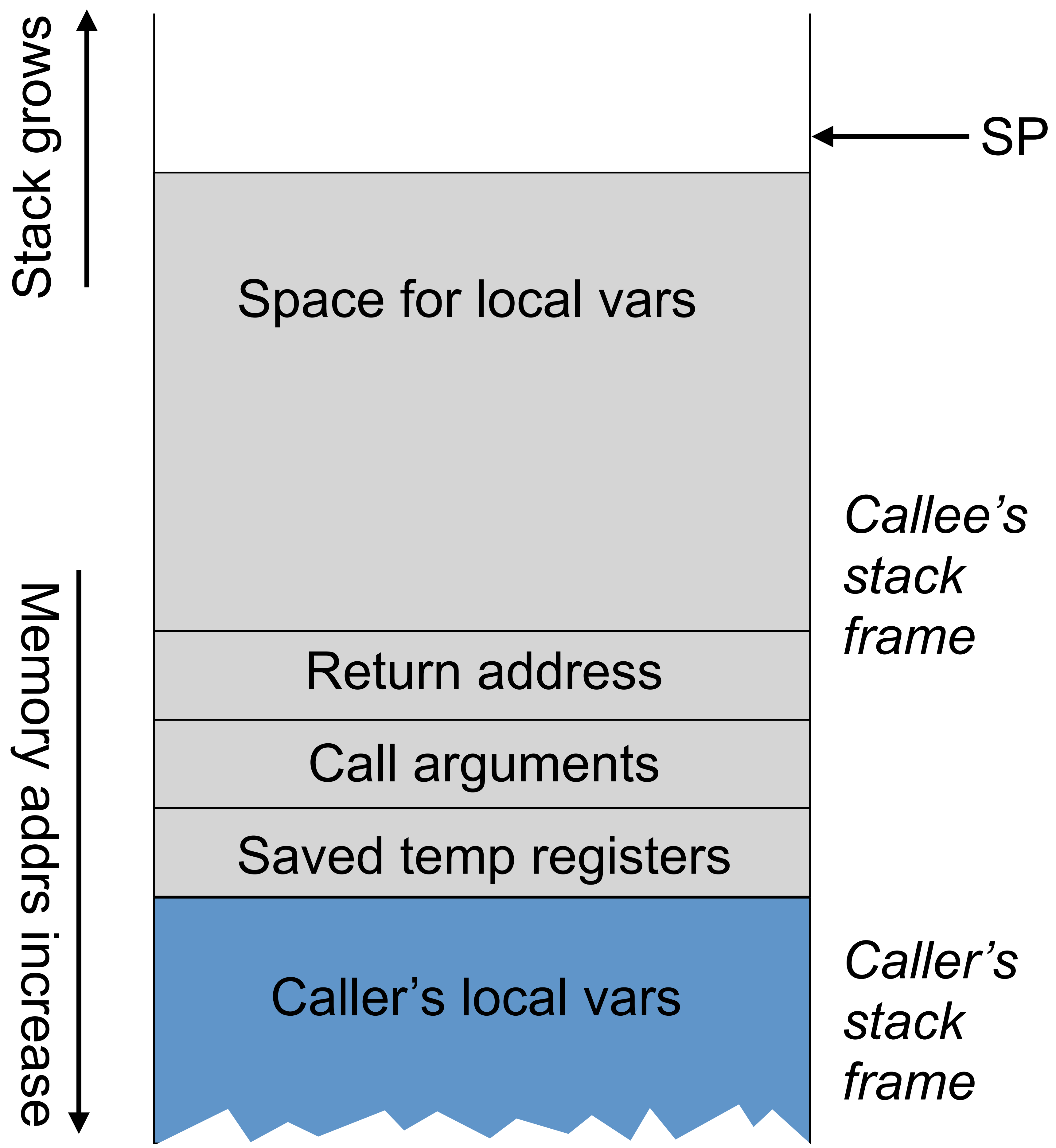
- Layer = group of modules
  - *Internet transport layer = UDP + TCP*
  - *Internet network layer = IP*
- Module communicates with modules in layer above/below, on the same layer stack instance, through API
  - *send/receive calls/notifications*
- Module communicates with modules in the same layer stack,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



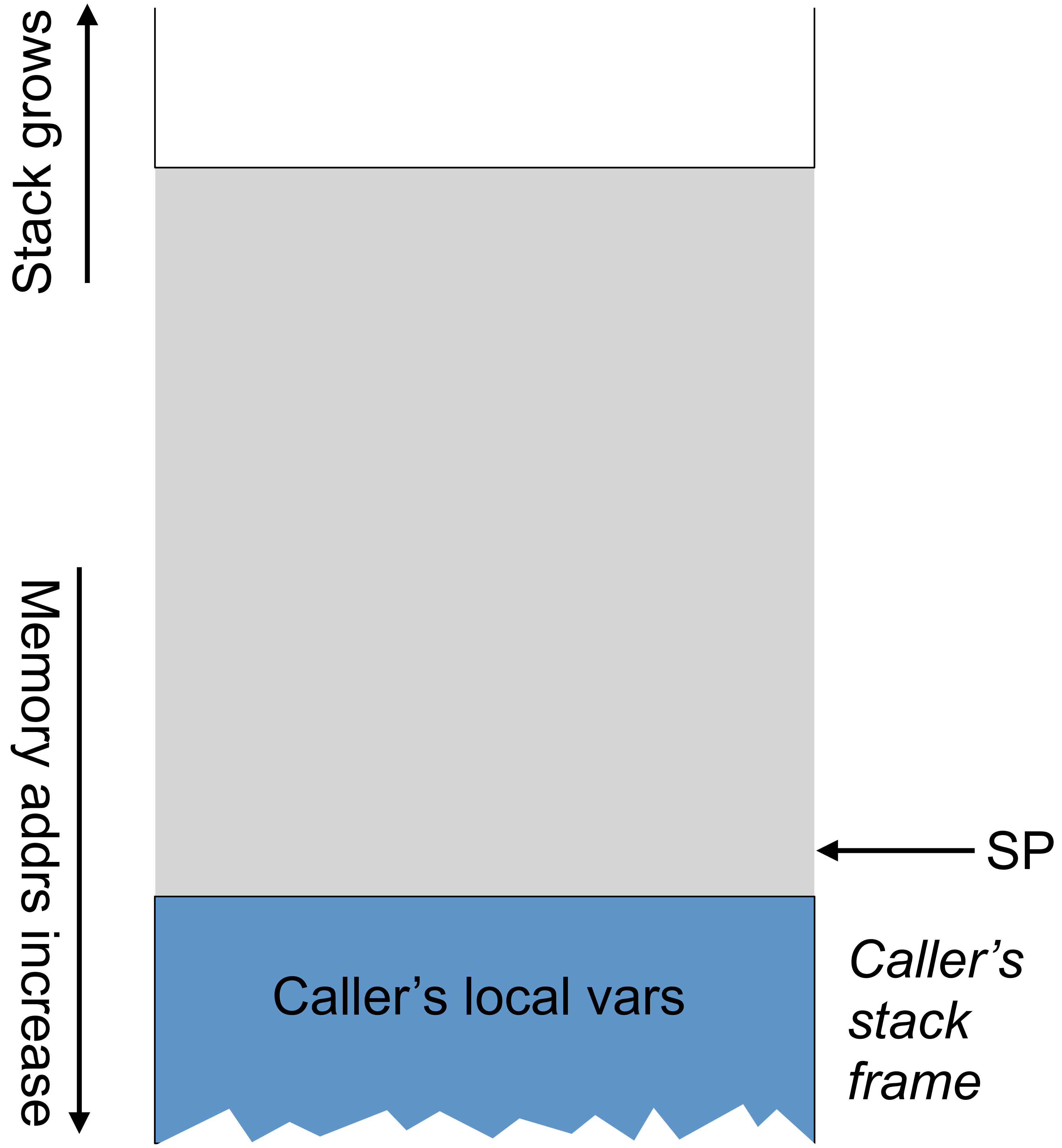
Stack-based calling convention



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*
  - *…*

Stronger intra-program modularity

- Untyped languages
- Weakly typed languages
  - *Have types, but can change (e.g., explicitly cast data from one type to another)*
- Strongly typed languages
  - *Each chunk of memory has well defined type*
- Ensuring type safety
  - *Static vs. dynamic*

Modularity violations

- *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*

Image courtesy of https://cs160debatable.weebly.com

Soft vs. enforced modularization

- Programmers are humans
  - *Trusting them is "soft" modularization*
  - *"Enforced" modularization: modules stay intact regardless of human mistakes*
- Better to trust compilers, runtimes, libraries, operating systems, …
  - *Widely used and robust (even though they too are buggy…)*
- Better to trust hardware
  - *Widely used and robust (even though it too is buggy…)*

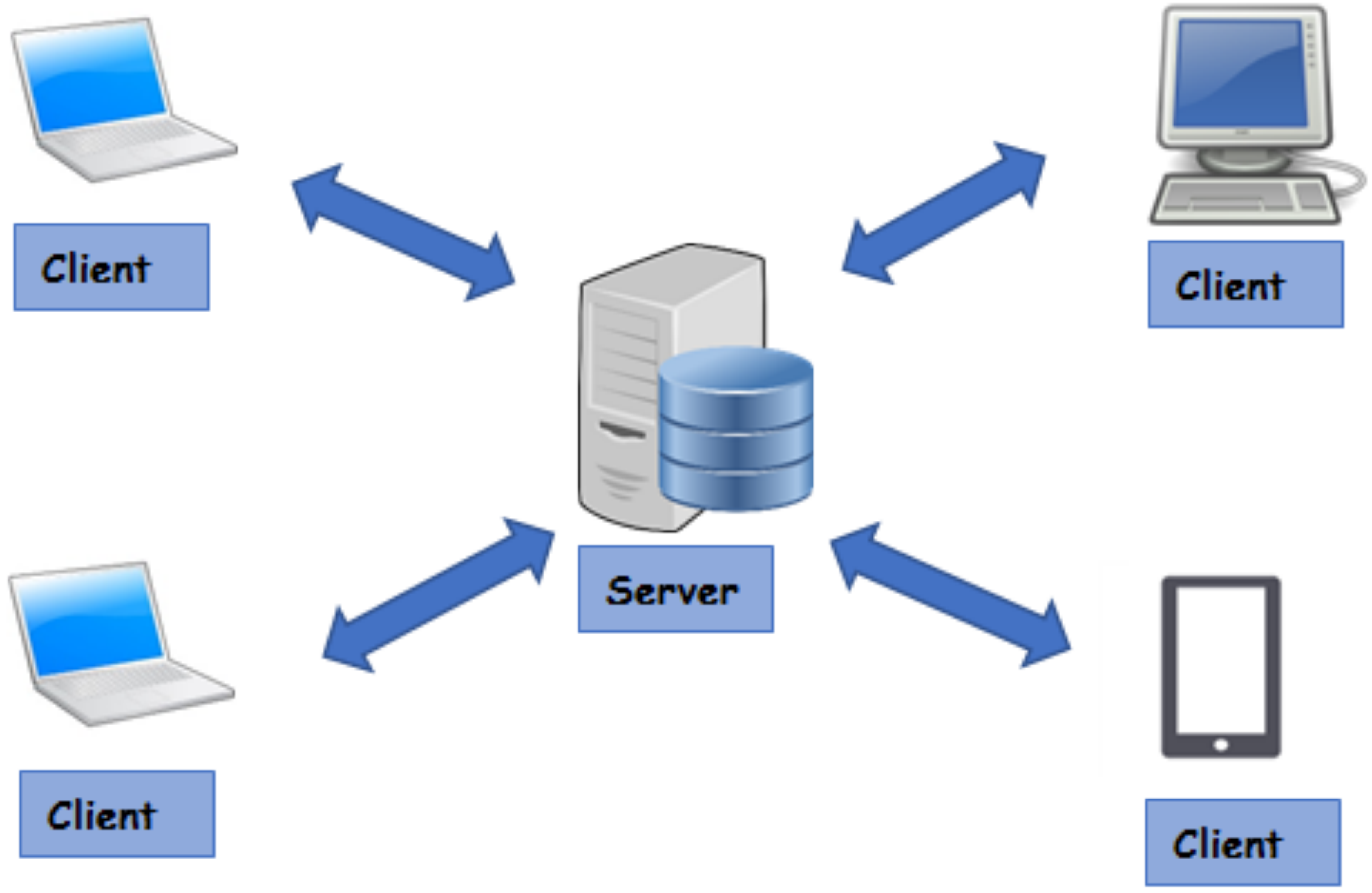
Client/Server Organization

Splitting into Clients and Servers

- Place modules in separate, strongly isolated domains, and have them communicate via messages
- Messages typically need to be marshalled/unmarshalled for send/receive
- Examples
  - *Web servers with clients connecting from remote machines*
  - *Front-end servers ↔ back-end servers*
  - *Microservices*
- Fate sharing

Physical (and virtual) servers

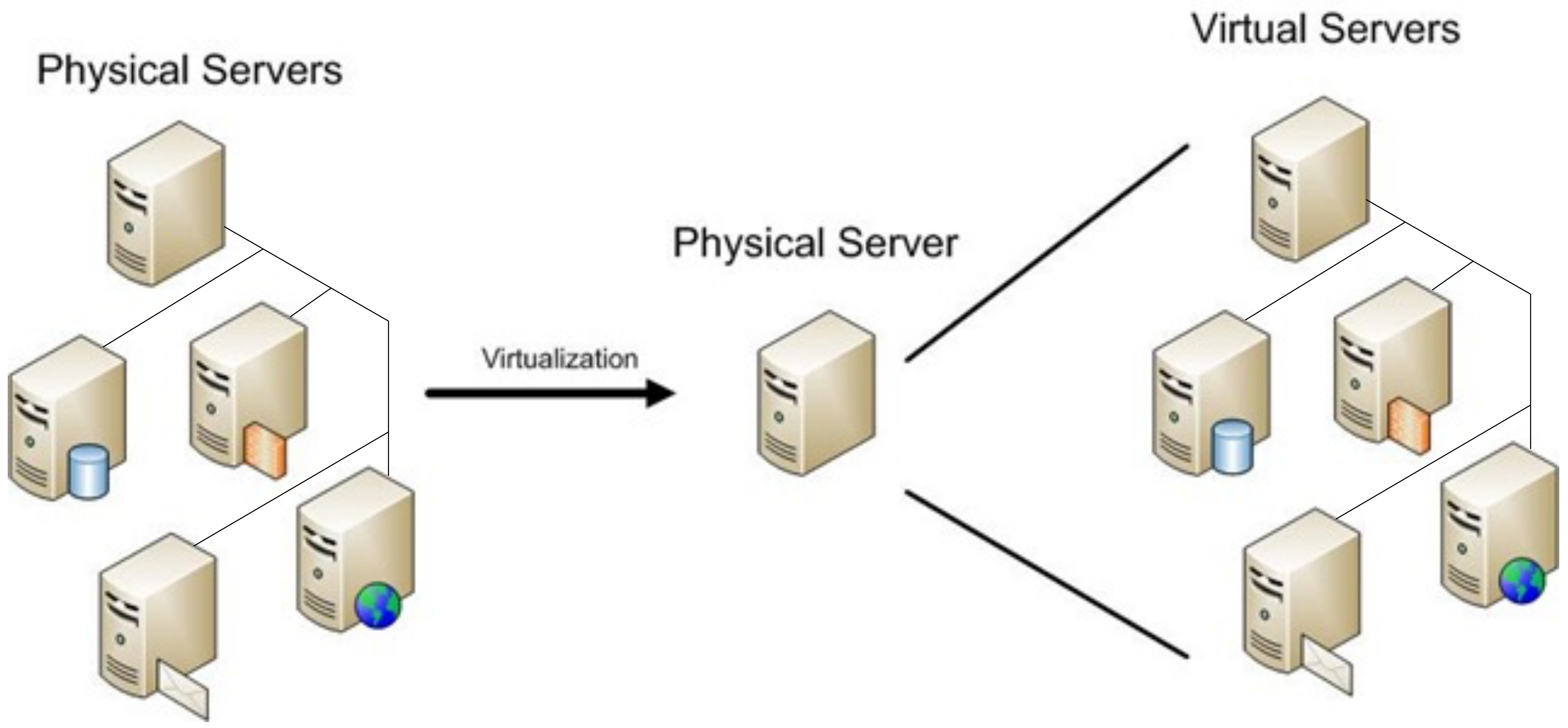
- Rely on physics
- Reduce fate sharing
- Improve encapsulation



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

Physical (and virtual) servers

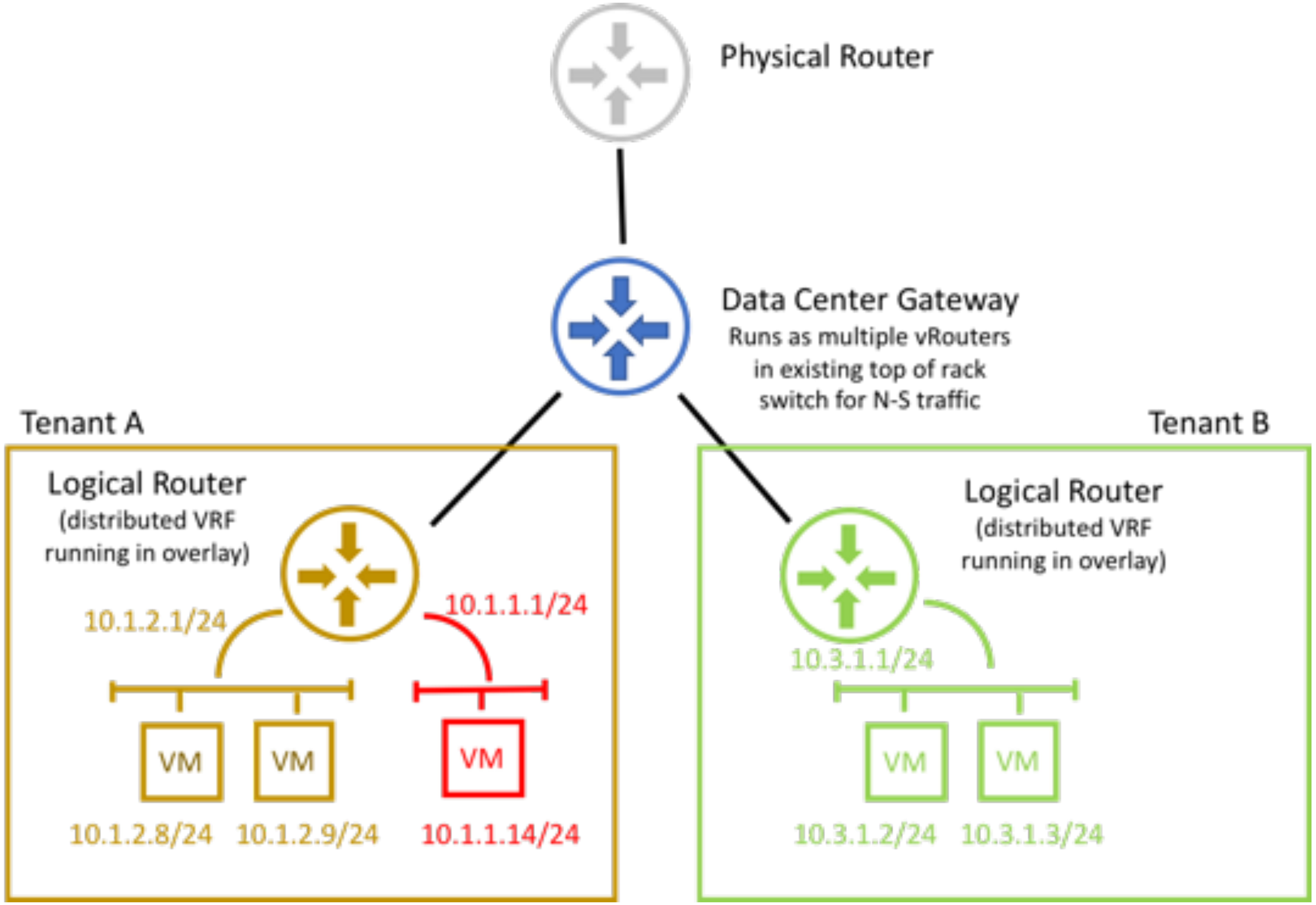
- Rely on physics
- Reduce fate sharing
- Improve encapsulation



https://rteb.files.wordpress.com/2009/08/consolidatinquick.jpg

Physical (and virtual) servers

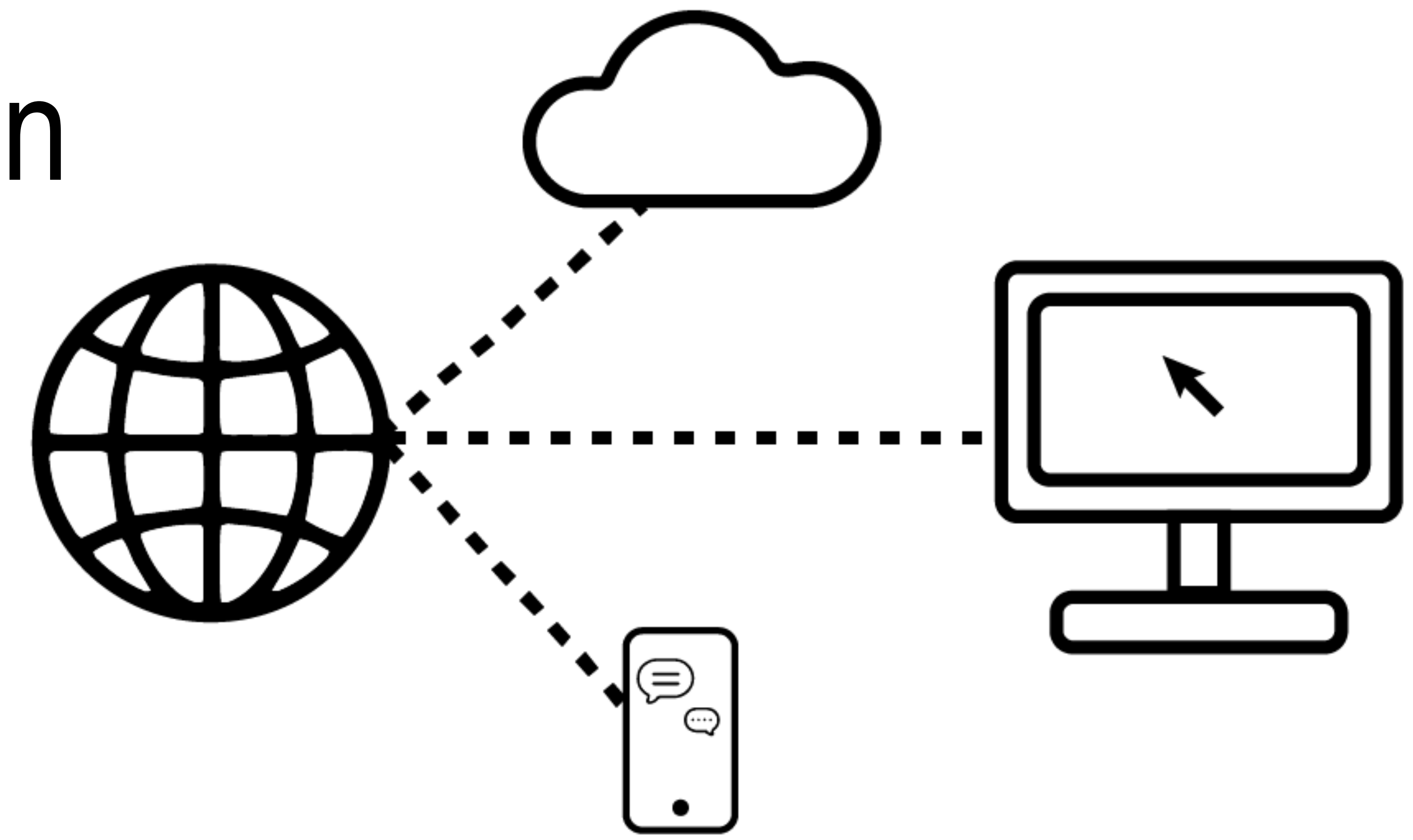
- Rely on physics
- Reduce fate sharing
- Improve encapsulation

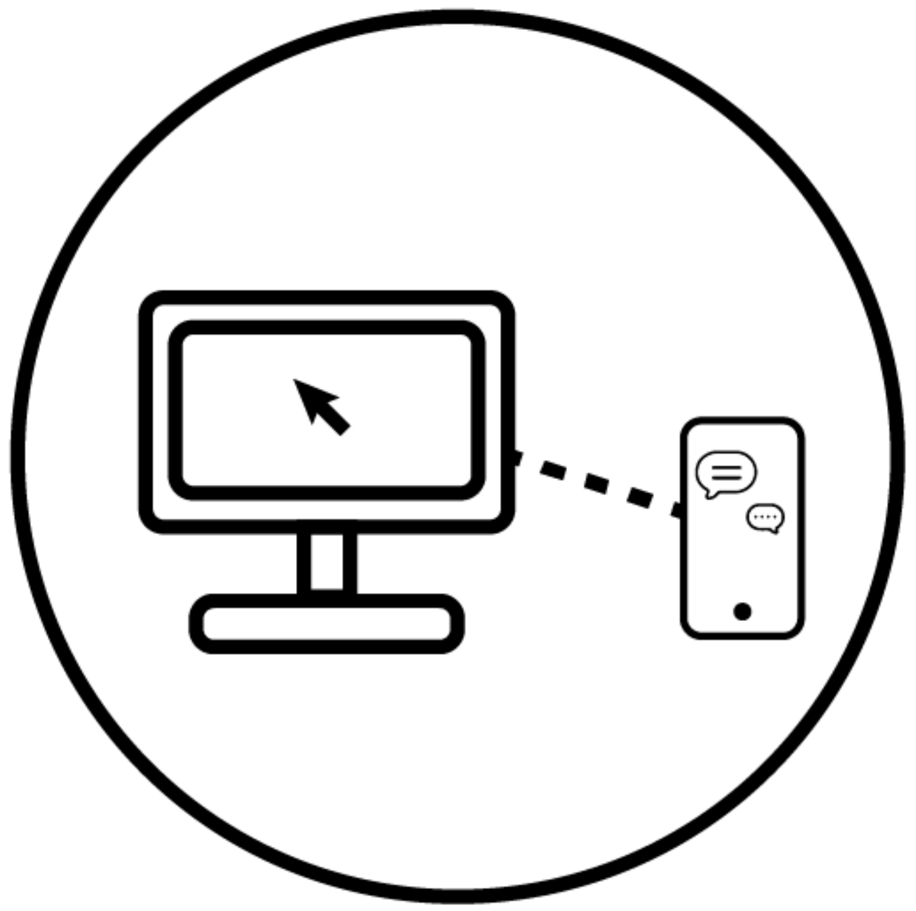


https://www.pluribusnetworks.com/blog/what-is-network-segmentation/

Physical (and virtual) servers

- Rely on physics
- Reduce fate sharing
- Improve encapsulation





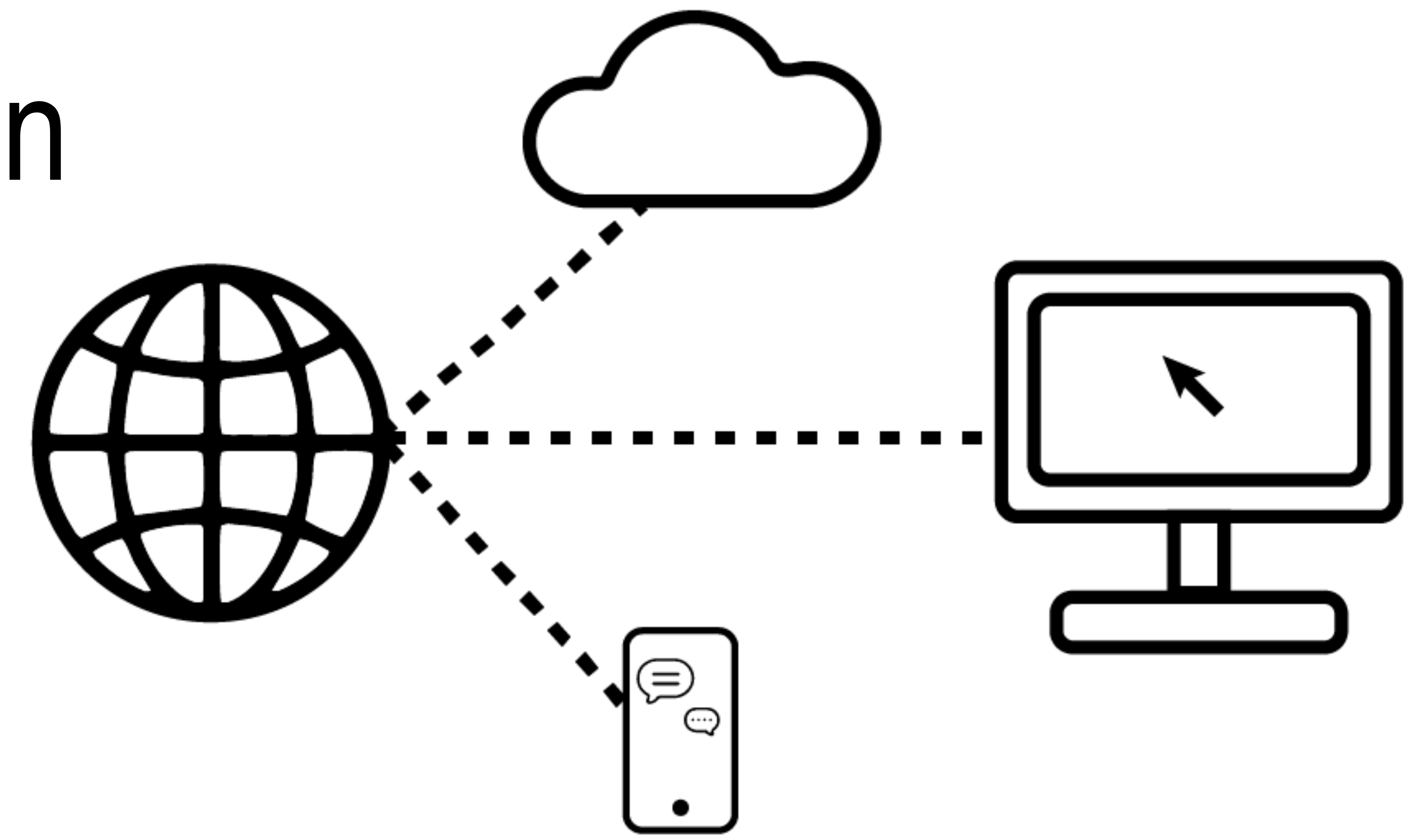
Air-gapped Network

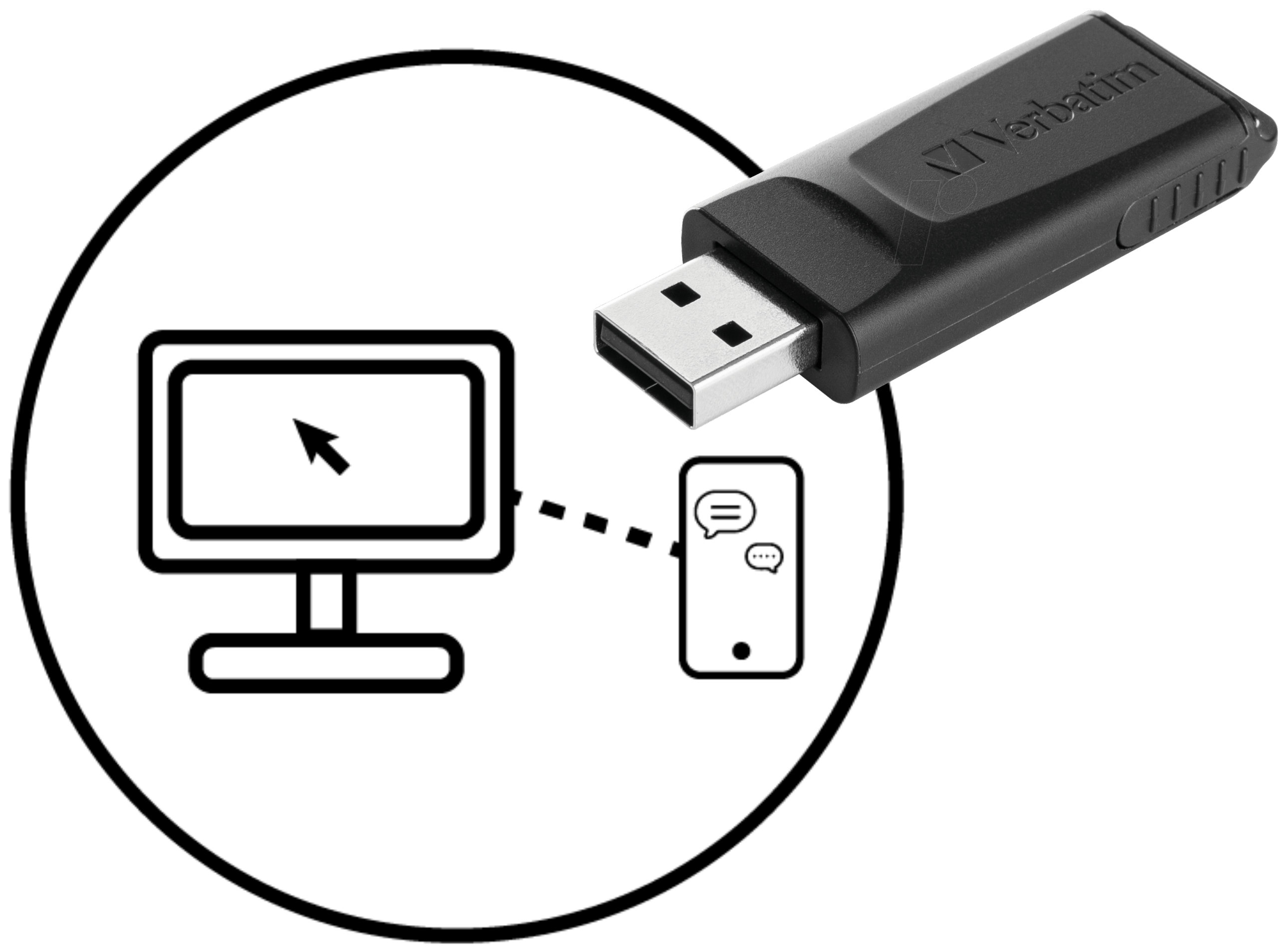
Devices included in the air-gapped network are physically isolated and can communicate with each other, but cannot communicate with any other network outside of the air-gap.

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

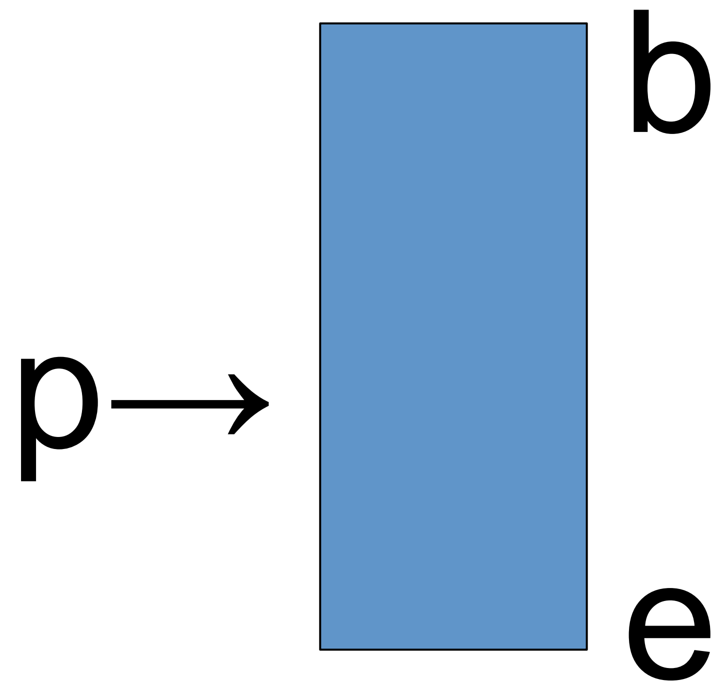
- An exercise in modularization of otherwise monolithic kernels
  - *Liedtke's minimality principle*
- Servers = trusted intermediaries
  - *Essentially daemon programs with some extra privileges*
  - *e.g., can access physical memory that would otherwise be off-limits*
- Talks to servers over IPC (inter-process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

Memory Safety

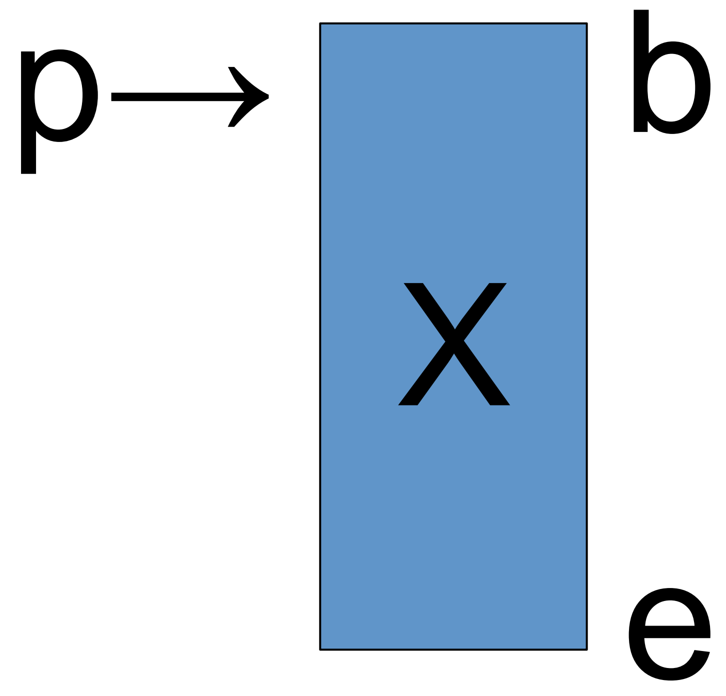
- Memory can be defined (allocated) or undefined (not allocated)
  - *Assume deallocated memory is never reused*
- Pointer is a capability **(p,b,e)**
  - *Base **b**, extent **e**, pointer **p***
- ***p** is safe iff it accesses memory within the target obj that **p** is based on
- <ins>An execution is memory-safe</ins> <=> all ptr derefs in that exec are safe
- <ins>A program is memory-safe</ins> <=> 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 <u>based on</u> 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., sub-object of **X**), or*
4. *the result of a computation involving operands that are ptrs based on **X** or non-ptrs*
    - copy of another pointer
    - pointer arithmetic
    - array indexing

An execution is memory-safe <=> object X is only accessed through pointers that are *based on* X

Memory Safety (recap)

- 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*

  b <= **p** <= e
- <u>An execution is memory-safe</u> <=>
  all pointer dereferences in that execution are safe
- <u>A program is memory-safe</u> <=>
  all possible executions (for all possible inputs) are memory-safe

* and that memory is defined

Benefits of Client/Server

- Narrow channels for error propagation
  - *Isolation between "caller" and "callee"*
  - *Memory safety introduces discipline in the access to memory objects*
- Decoupling
  - *Can fail independently —> the opposite of "fate sharing"*
  - *Rely on timeouts to infer remote failure*
- Forcing function to documenting 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*

Remote Procedure Calls (RPC)

Mechanics of RPC

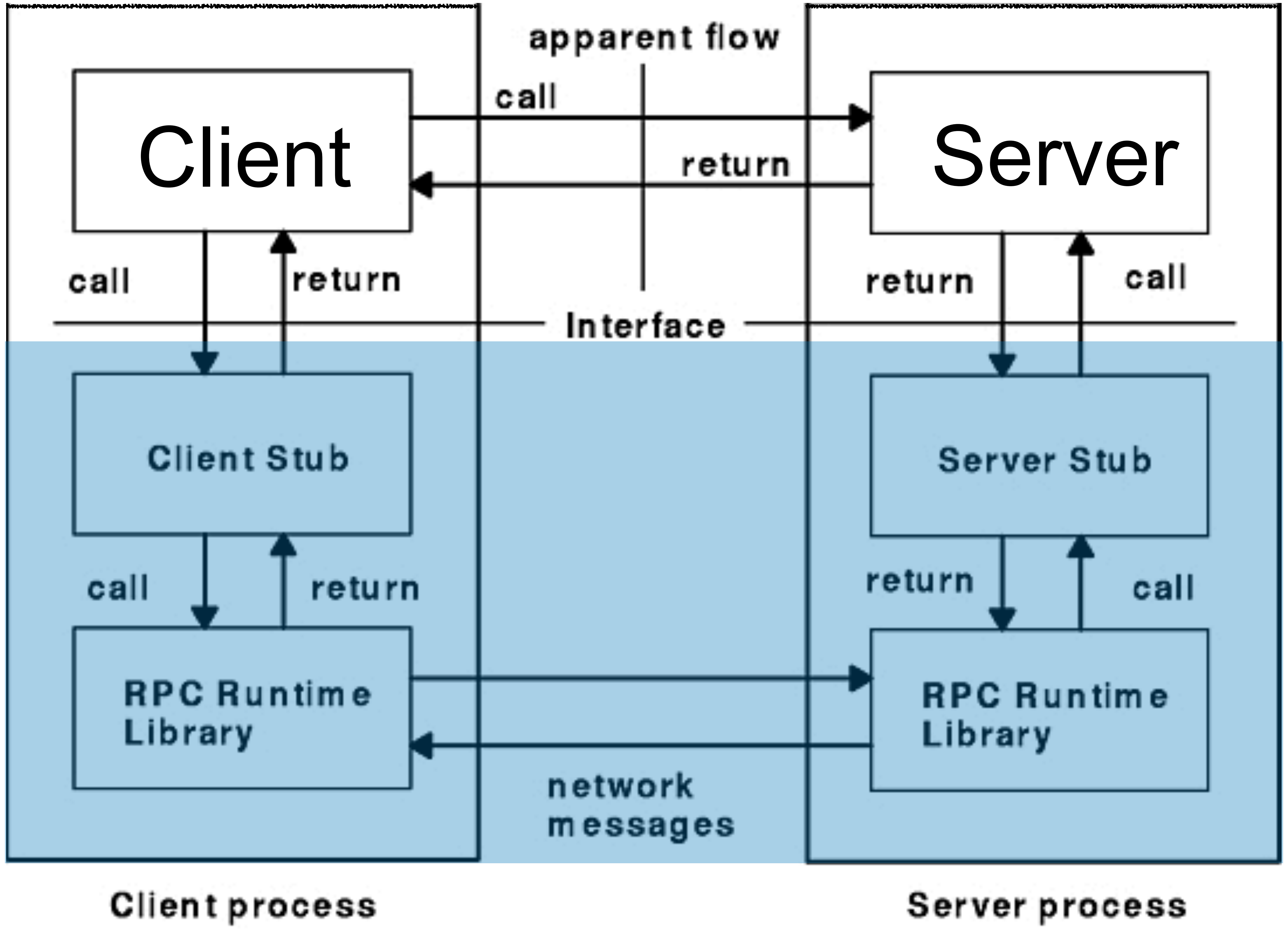


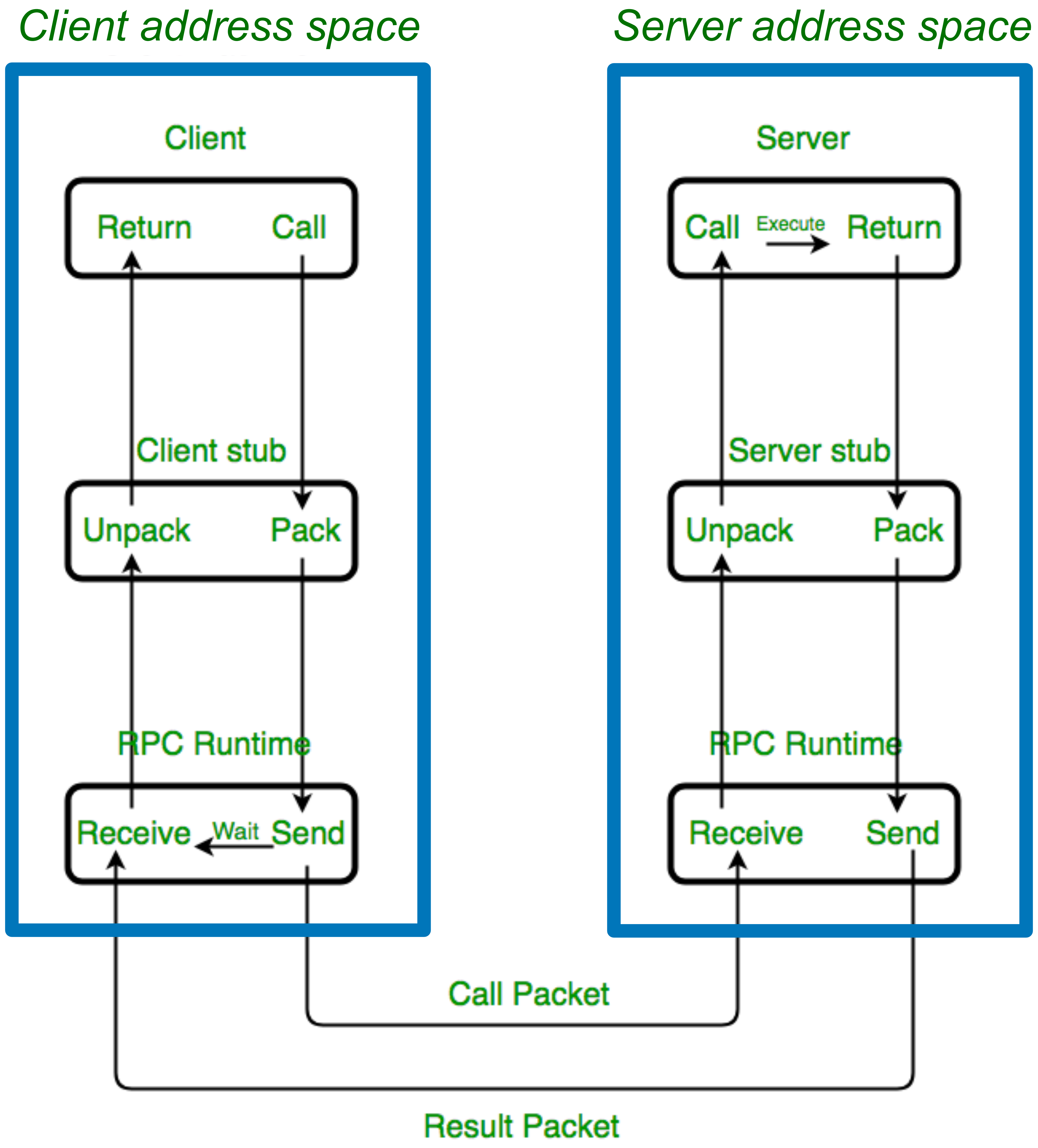
Image courtesy of https://www.ibm.com/support/knowledgecenter

Mechanics of RPC

Local procedure call

Parameters -> message

Message -> packet



Local procedure invocation

Message -> parameters

Packet -> message

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)

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 ::tutorial::Person_PhoneNumber& phones(int index) const;
inline ::tutorial::Person_PhoneNumber* mutable_phones(int index);
inline ::tutorial::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

→ protoc --cpp_out=$DST_DIR contacts.proto → contacts.pb.h
contacts.pb.cc

```
// 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)

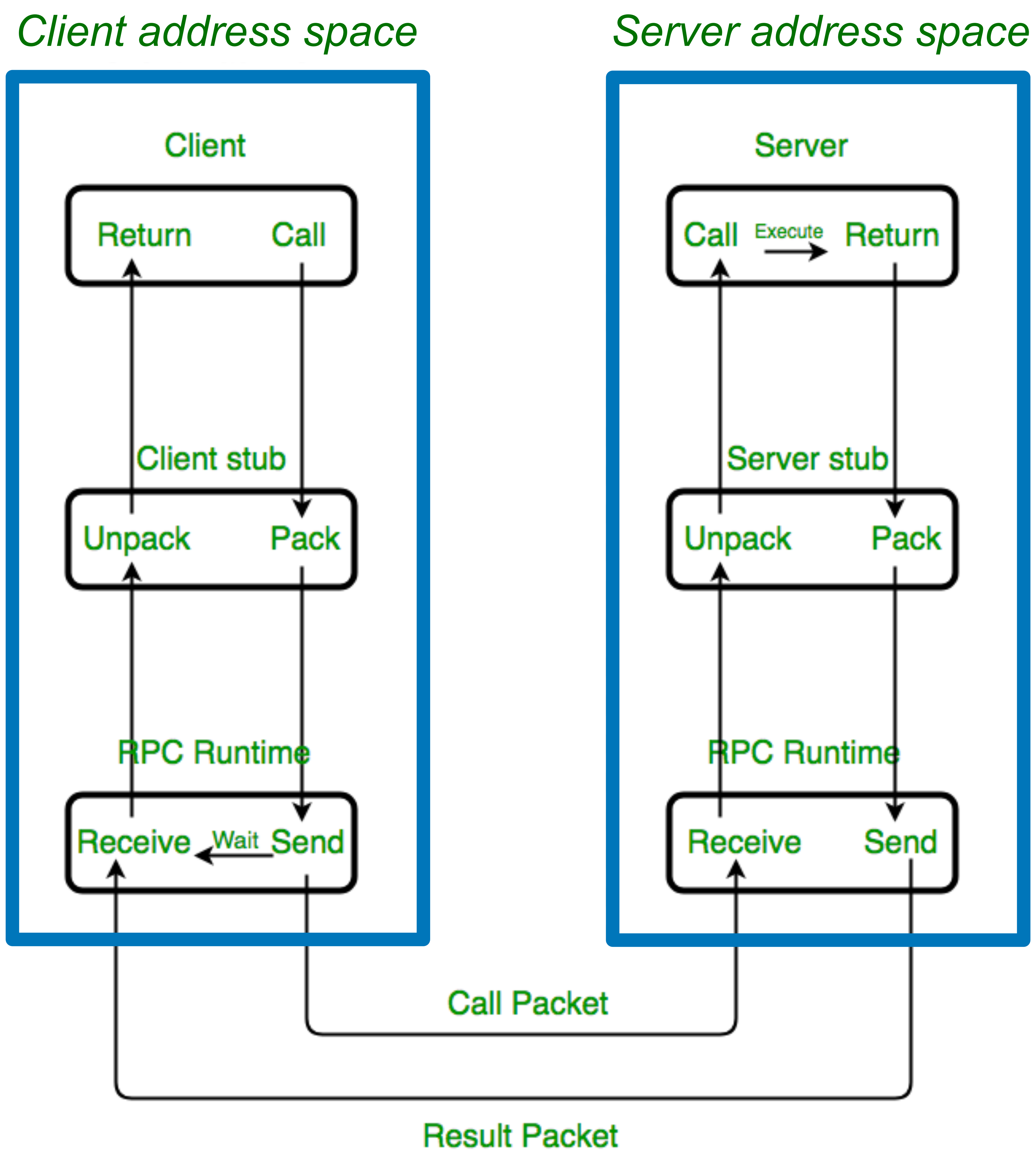
```
// Interface exported by the server.
service Contacts {
  // A simple RPC.
  //
  // Obtains the feature of a given Person.
  rpc GetNumber(Person) returns (PhoneNumber) {}

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

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

contacts.proto

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

Summary



- Define the service in an IDL file (.proto)
- 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

Benefits

- 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

- 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

Other forms of message-based interaction

- Push notifications (instead of pull)
- Publish/subscribe

Outline

- Brief recap
- Client/Server Organization: Overview
- Remote Procedure Calls (RPC)