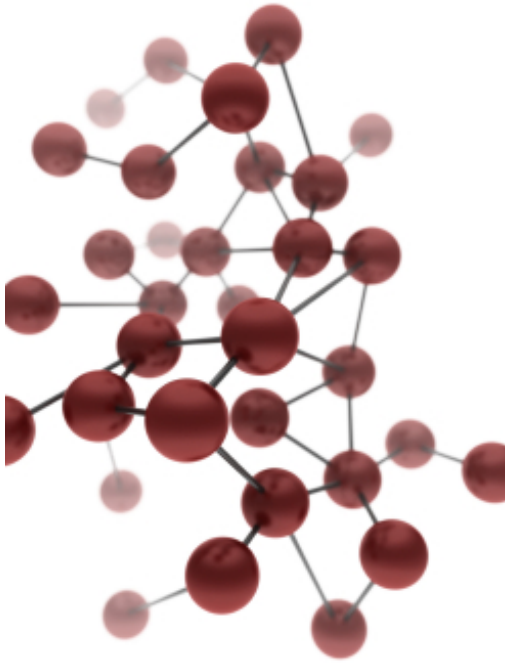


DNS and RPC
Lecture 10

Today

- General info on midterm
- Finishing up DNS
- Server Calls
- Remote Procedure Calls (RPC)

Midterm

- 11/12 in class
- 1-page notes allowed
- Covering all the materials until today (11/7)

Sample Midterm Questions

- Why do we need CIDR?

- Which of the following mechanisms does TCP use to avoid overloading the receiver? (circle one)
  - Flow control
  - Congestion control
  - None of the above

- Describe two methods to attack DNS

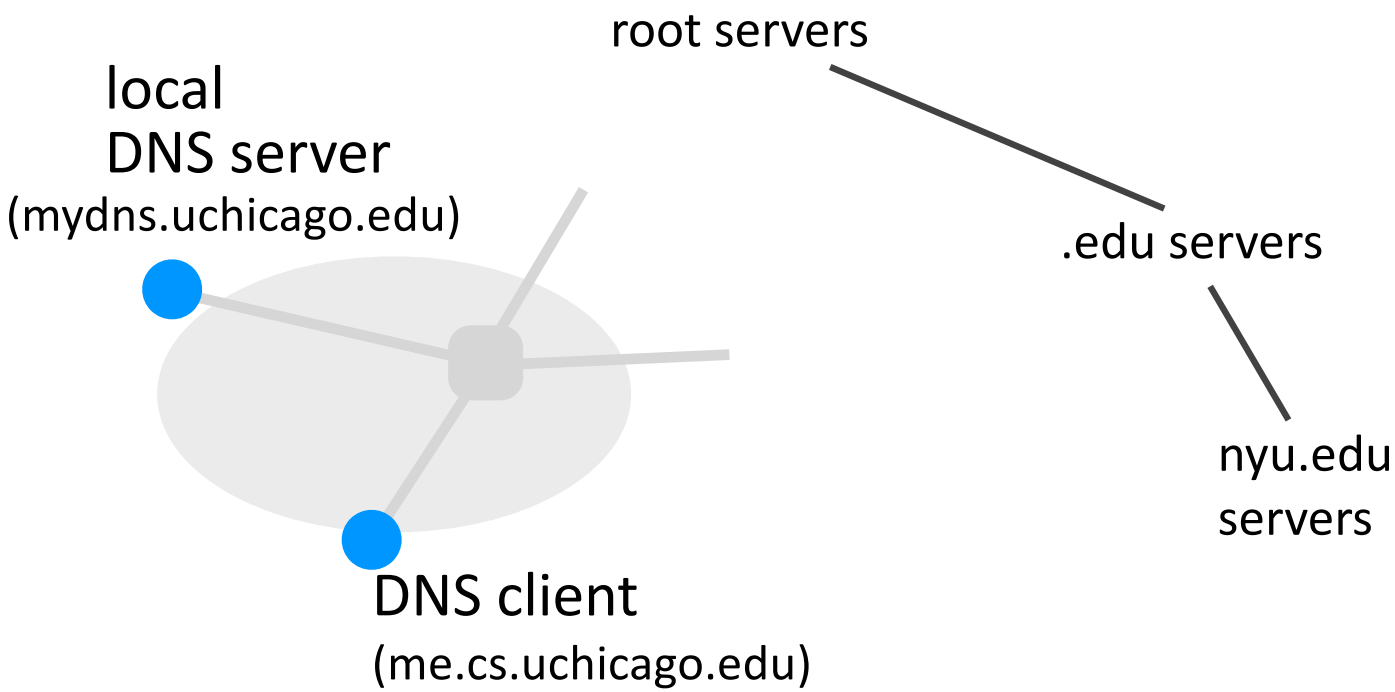
- TCP vs. UDP : which of the following is true?
  - TCP is stateless, UDP is stateful
  - TCP is stateful, UDP is stateless

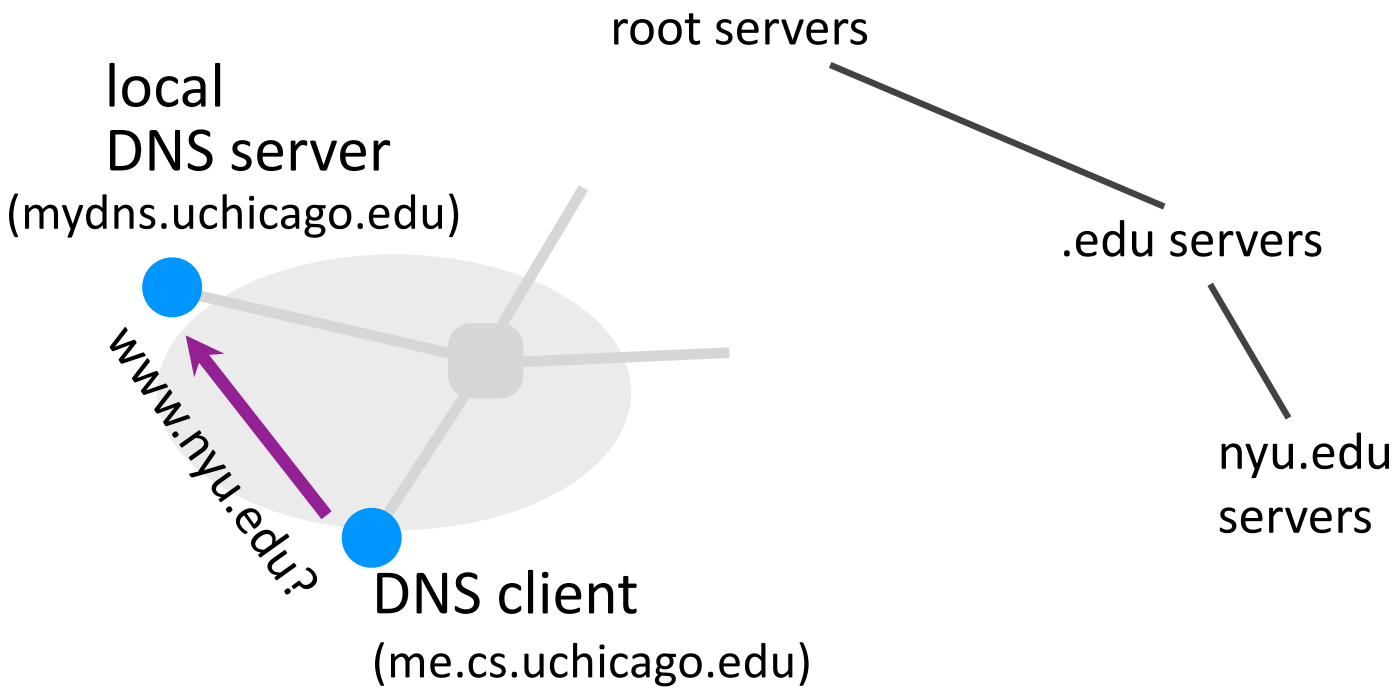
Using DNS (Client/App View)

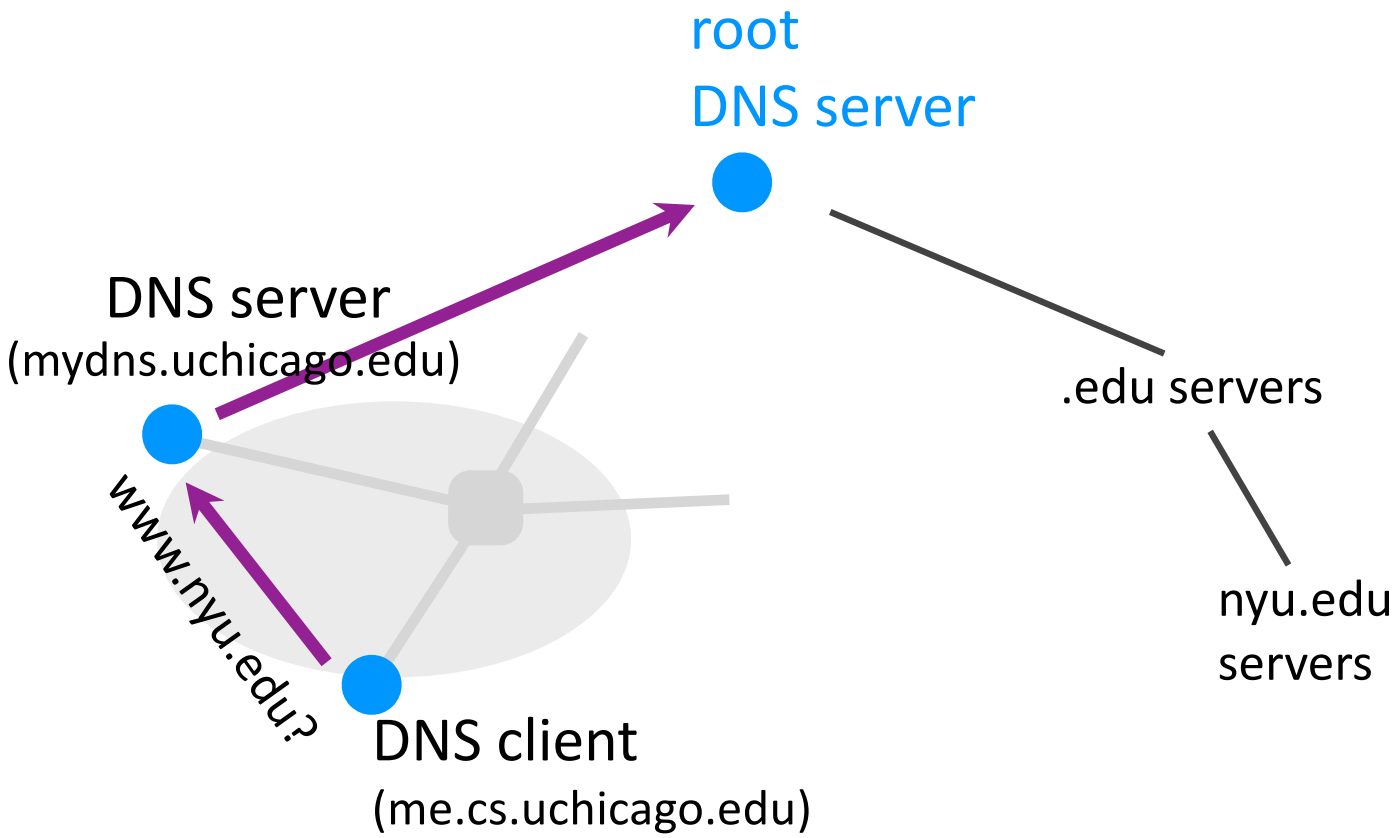
- Two components
  - Local DNS servers
  - Resolver software on hosts

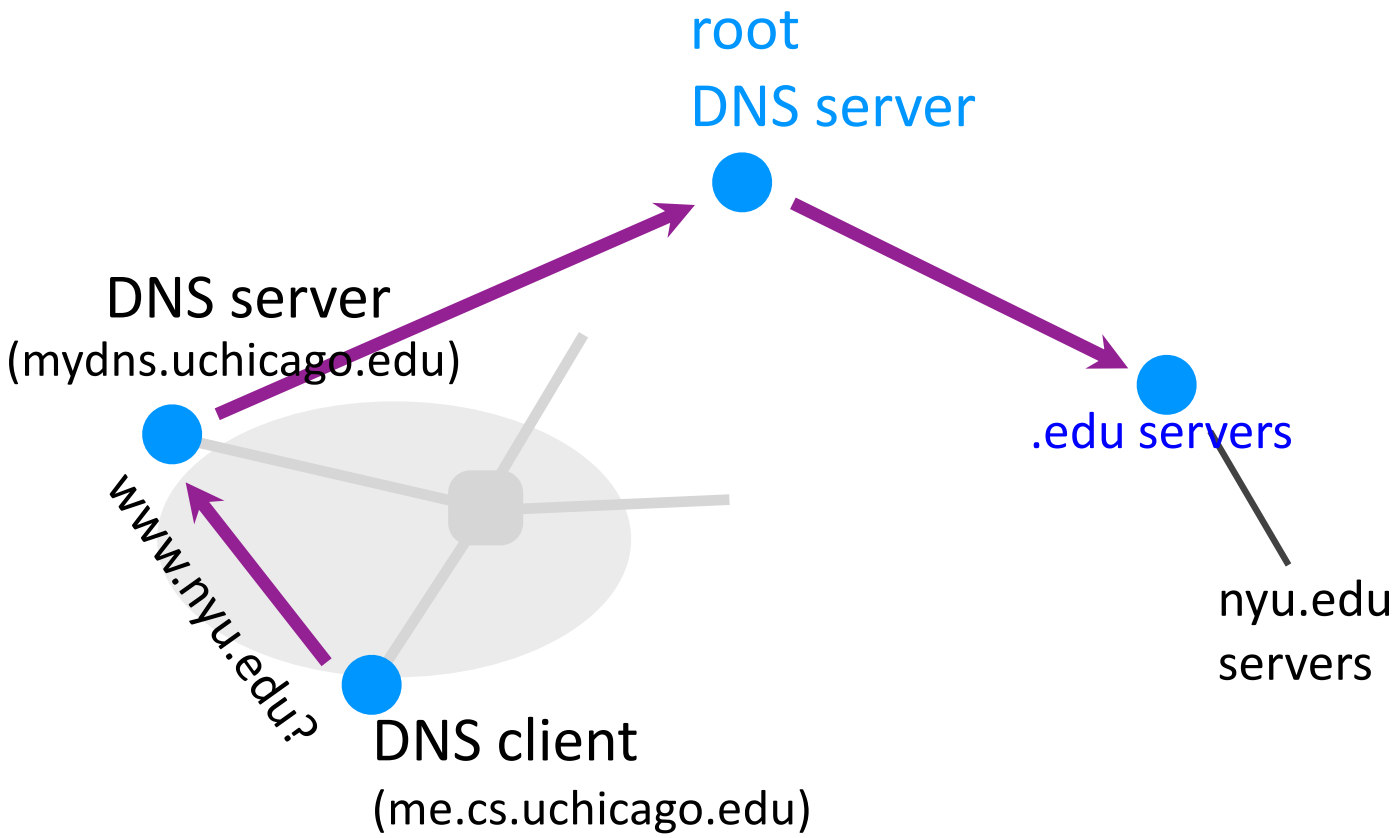
- Local DNS server ("default name server")
  - Clients configured with the default server's address or learn it via a host configuration protocol (e.g., DHCP – future lecture)

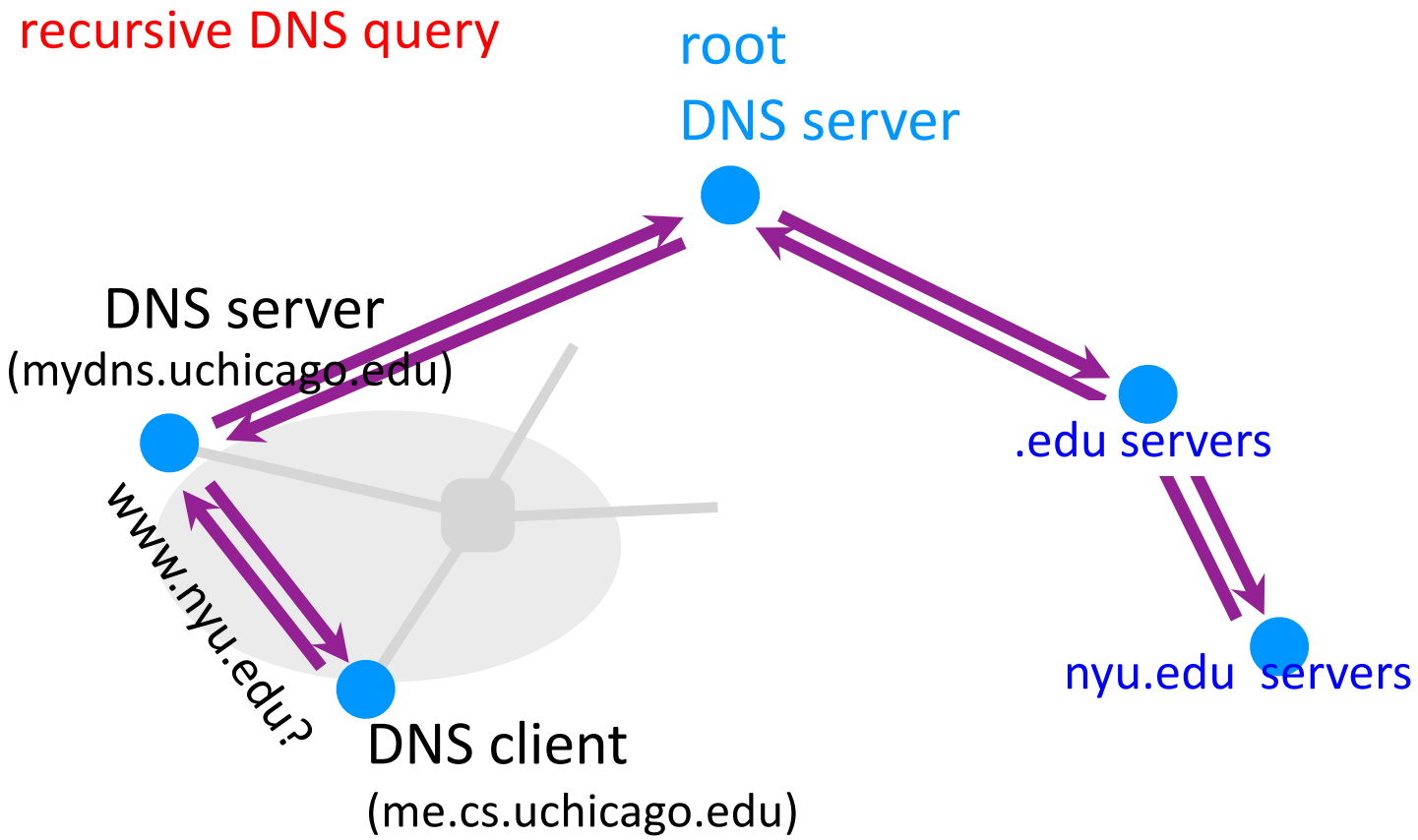
- Client application
  - Obtain DNS name (e.g., from URL)
  - Triggers DNS request to its local DNS server

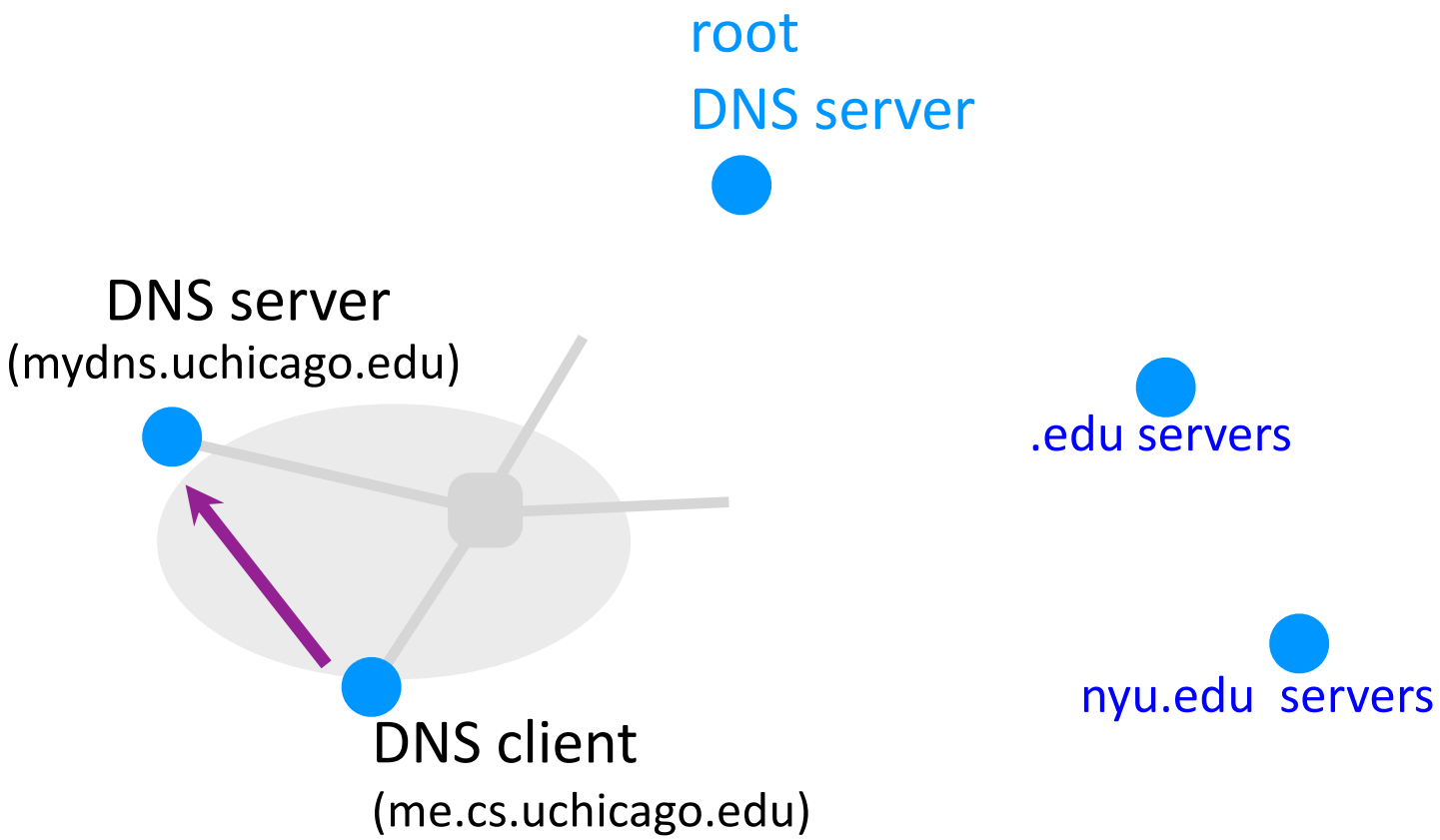

root servers
local
DNS server
(mydns.uchicago.edu)
.edu servers
nyu.edu
servers
DNS client
(me.cs.uchicago.edu)

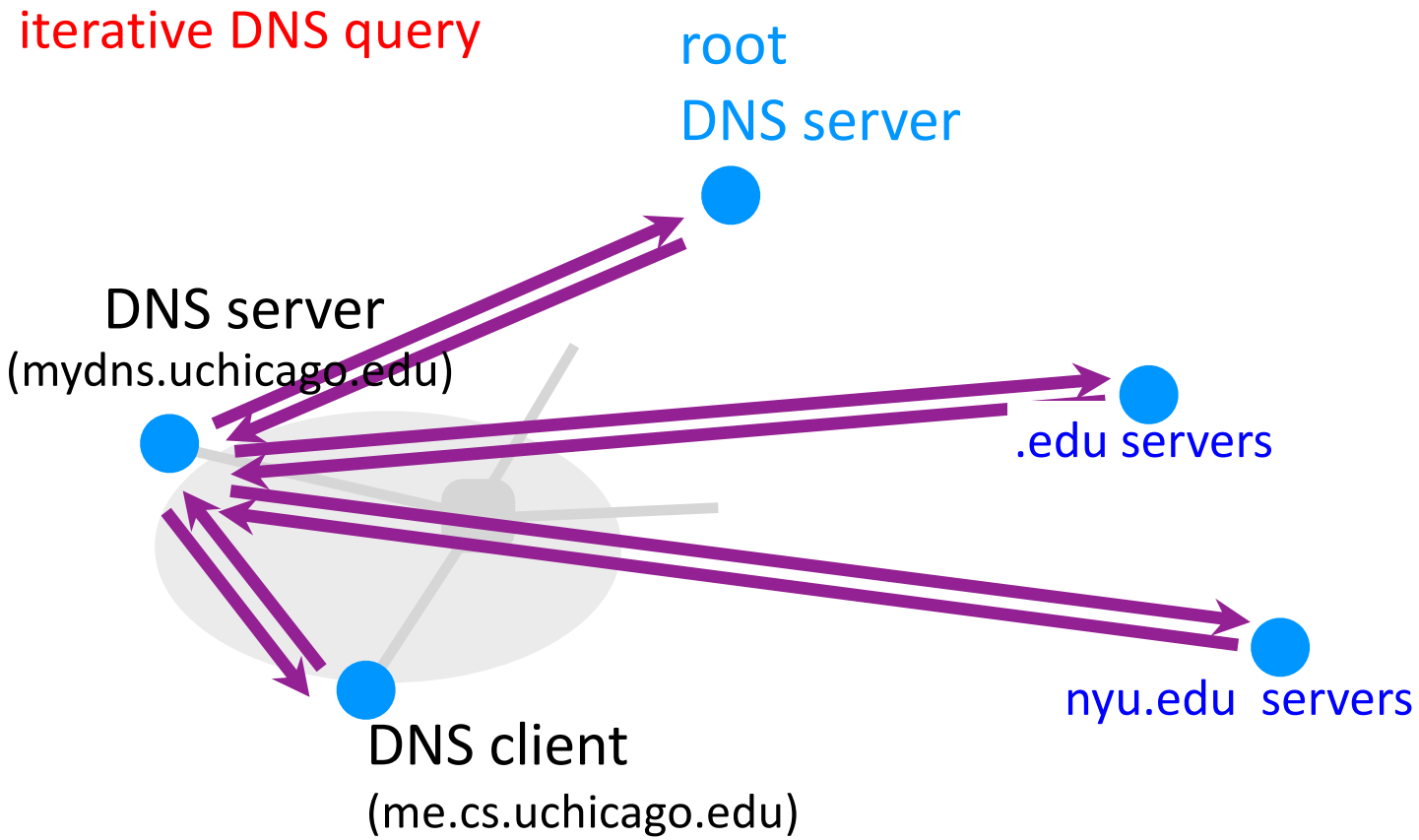

root servers
local
DNS server
(mydns.uchicago.edu)
.edu servers
www.nyu.edu?
nyu.edu
servers
DNS client
(me.cs.uchicago.edu)


root
DNS server
DNS server
(mydns.uchicago.edu)
.edu servers
www.nyu.edu?
nyu.edu
servers
DNS client
(me.cs.uchicago.edu)


root
DNS server
DNS server
(mydns.uchicago.edu)
.edu servers
www.nyu.edu?
nyu.edu
servers
DNS client
(me.cs.uchicago.edu)


recursive DNS query
root
DNS server
DNS server
(mydns.uchicago.edu)
.edu servers
www.nyu.edu?
nyu.edu servers
DNS client
(me.cs.uchicago.edu)


root
DNS server
DNS server
(mydns.uchicago.edu)
.edu servers
nyu.edu servers
DNS client
(me.cs.uchicago.edu)


iterative DNS query
root
DNS server
DNS server
(mydns.uchicago.edu)
.edu servers
nyu.edu servers
DNS client
(me.cs.uchicago.edu)

DNS Protocol

- Query and Reply messages; both with the same message format
  - *We're skipping the details here*
- Client-Server interaction on UDP Port 53
  - Spec supports TCP too, but not always implemented

Goals – how are we doing?

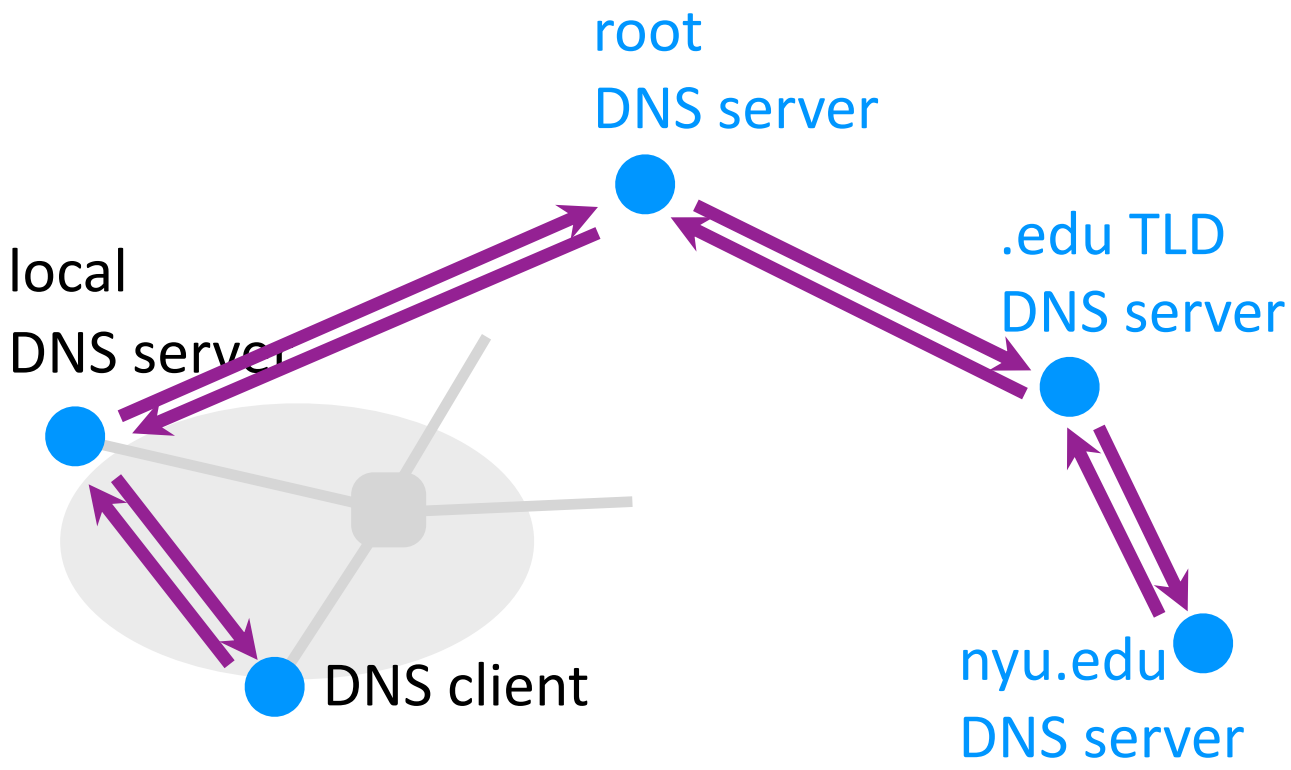
- Scalable
  - many names
  - many updates
  - many users creating names
  - many users looking up names
- Highly available

Per-domain availability

- DNS servers are replicated
  - Primary and secondary name servers required
  - Name service available if at least one replica is up
  - Queries can be load-balanced between replicas
- Try alternate servers on timeout
  - Exponential backoff when retrying same server

Goals – how are we doing?

- Scalable
  - many names
  - many updates
  - many users creating names
  - many users looking up names
- Highly available
- Correct
  - no naming conflicts (uniqueness)
  - consistency
- Lookups are fast


root
DNS server
.edu TLD
DNS server
local
DNS server
DNS client
nyu.edu
DNS server

Caching

- Caching of DNS responses at all levels

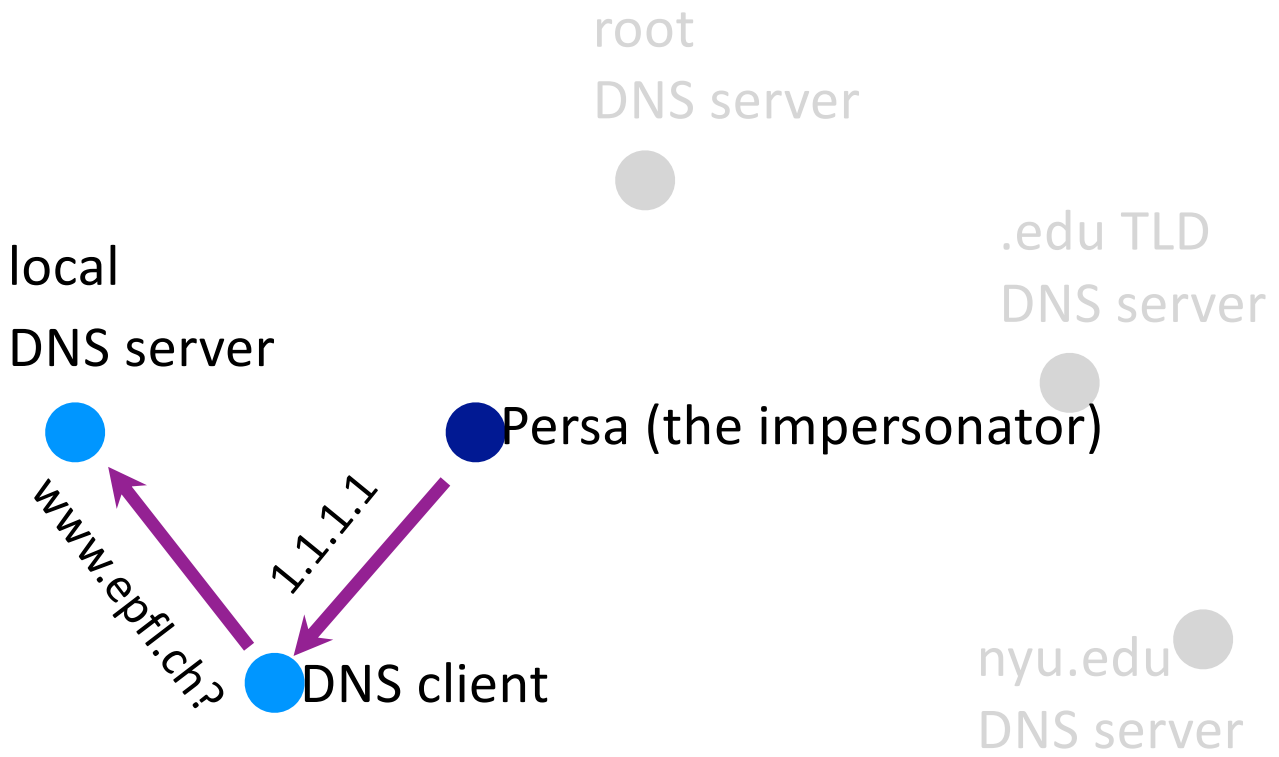
- Reduces load at all levels
- Reduces delay experienced by DNS client

DNS Caching

- How DNS caching works
  - DNS servers cache responses to queries
  - Responses include a "time to live" (TTL) field
  - Server deletes cached entry after TTL expires

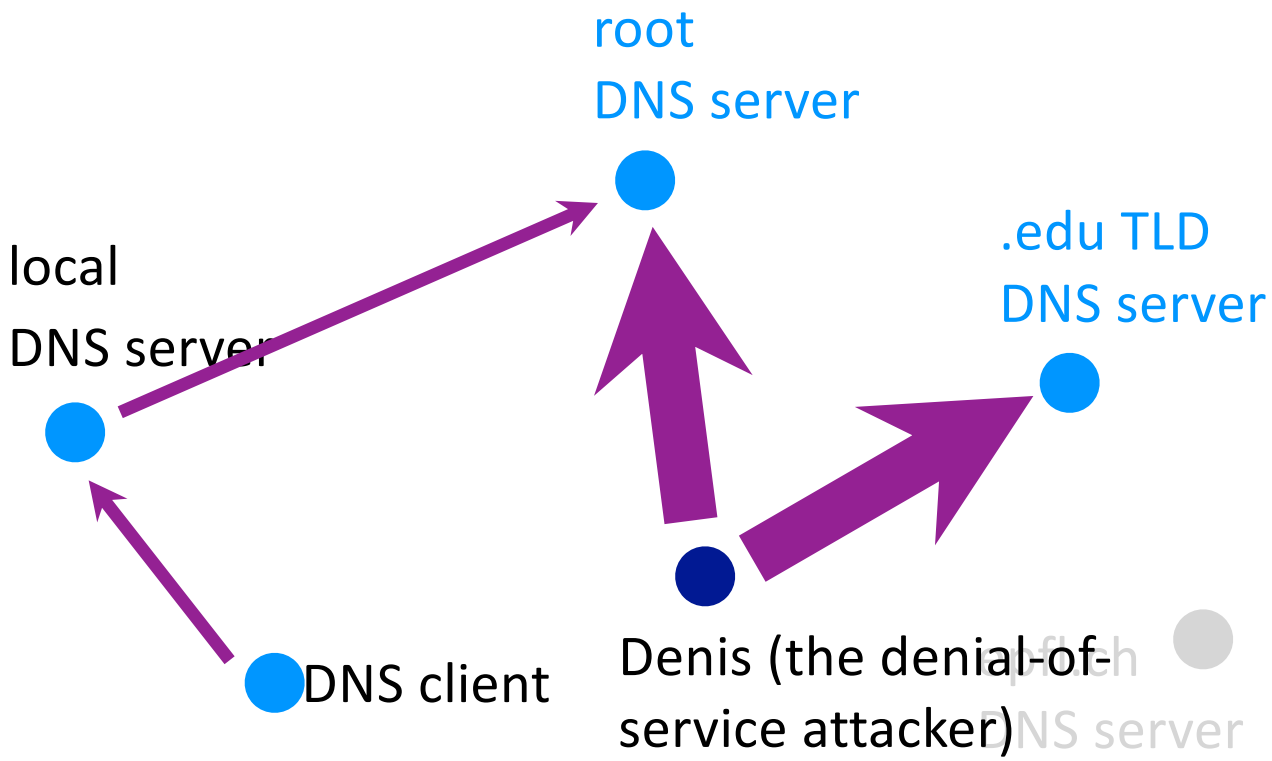
- Why caching is effective
  - The top-level servers very rarely change
  - Popular sites visited often → local DNS server often has the information cached

How can one attack DNS?


root
DNS server
.edu TLD
DNS server
local
DNS server
Persa (the impersonator)
www.epfl.ch?
1.1.1.1
DNS client
nyu.edu
DNS server

How can one attack DNS?

- Impersonate the local DNS server
  - *give the wrong IP address to the DNS client*


root
DNS server
.edu TLD
DNS server
local
DNS server
DNS client
Denis (the denial-of-
service attacker)

How can one attack DNS?

- Impersonate the local DNS server
  - *give the wrong IP address to the DNS client*
- Denial-of-service the root or TLD servers
  - *make them unavailable to the rest of the world*

Important Properties of DNS

- Administrative delegation and hierarchy results in:
- Easy unique naming
- “Fate sharing” for network failures
- Reasonable trust model
- Caching lends scalability, performance

DNS provides Indirection

- Addresses can change underneath
  - Move www.cnn.com to a new IP address
  - Humans/apps are unaffected
- Name could map to multiple IP addresses
  - Enables load-balancing
- Multiple names for the same address
  - *E.g.*, many services (mail, www, ftp) on same machine
- Allowing “host” names to evolve into “service” names

Now Back to Server Design

(Project 2 deals with Client Design)

Server Designs

- Iterative vs. Concurrent
  - Iterative are simple and inefficient
  - Concurrent are complex and efficient
- Connection-Oriented vs. Connectionless
  - Semantics of TCP and UDP are very different
  - Need to evaluate pros/cons
    - Connectionless more flexible and lightweight
    - Connection-oriented provides reliability but may be more complex
- Stateful vs. Stateless
  - Keeping state is complex but sometime is necessary
  - Statefulness may affect scalability
  - Statelessness may affect correctness

Iterative, Connection-Oriented Server

1. Create a TCP socket (s1)
2. Bind it to a port/address (INADDR_ANY)
3. Place the socket in passive mode (listen)
4. Accept connection and obtain socket s2
5. Interact with client using socket s2
6. Close socket s2 and go to (4) or...
7. Exit

Server can only serve one client at a time.
It works with one TCP connection, then closes it and deals with the next TCP connection

Iterative, Connectionless Server

1. Create a UDP socket
2. Bind it to a port/address (INADDR_ANY)
3. Read a message from a client (recvfrom)
4. Process the request
5. Send reply to the client (sendto)
6. Go to (3) or...
7. Exit

Concurrent, Connectionless Server

1. Create a UDP socket
2. Bind it to a port/address (INADDR_ANY)
3. Read a message from a client (recvfrom)
4. Fork a slave (process/thread)
    1. Process the request
    2. Send reply to the client (sendto)
    3. Exit
5. Go to (3) or...
6. Exit

Concurrent, Connection-oriented Server

1. Create a TCP socket (s1)
2. Bind it to a port (INADDR_ANY)
3. Place the socket in passive mode (listen)
4. Accept connection (uses s1, returns new socket s2)
5. Fork slave process
    1. Close old socket (C-s1)
    2. Interact with the client using new socket (C-s2)
    3. Close new connection (C-s2)
    4. Exit
6. Close new socket (s2)
7. Go to (4) or...
8. Exit

One slave will not block another slave

But large overhead in opening and closing the slave process

TCP server with select() syscall

- Each process tells the OS: “please put me to sleep and wake me up when something interesting happens”
  - “something interesting” == a new TCP connection or some bytes sent on any existing TCP connection.
- Done via Select()

Concurrent, Connection-oriented Server

1. Create a TCP socket (s1) and insert it in pool of open connections
2. Bind it to a port (INADDR_ANY) and place the socket in passive mode (listen) Add s1 to (new) pool of connections
3. Apply select( ) to a pool of connections
4. If s1 has an event pending
   - Accept connection (returns new socket s2)
   - Add new socket to pool of open connections (s2)
5. else
   - Interact with the client associated with the socket with pending events (read and possibly write data)
   - Possibly close connection and remove socket from pool
6. Go to (3) or ...
7. Exit

Select

- **`int select(int n, fd_set *readfds, fd_set *writefds, fd_set *exceptfds, struct timeval *timeout)`**
  - **`readfds`**: set of descriptors to be checked for available characters. More precisely, checks if read doesn't block (a file descriptor is also ready on end-of-file)
  - **`writefds`**: set of descriptors to be checked to see if a write will not block
  - **`exceptfds`**: set of descriptors to be checked for exceptions
  - **`timeout`**: how long to wait for changed FD: null means infinite, if timeval all 0's, check but don't block
- On exit, the sets are modified in place to indicate which descriptors actually changed status

Remote Procedure Call

Remote Procedure Call

- Basic RPC operation

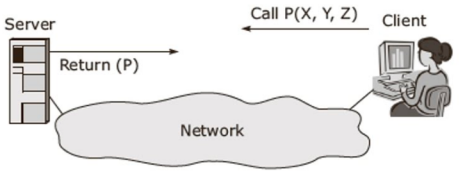


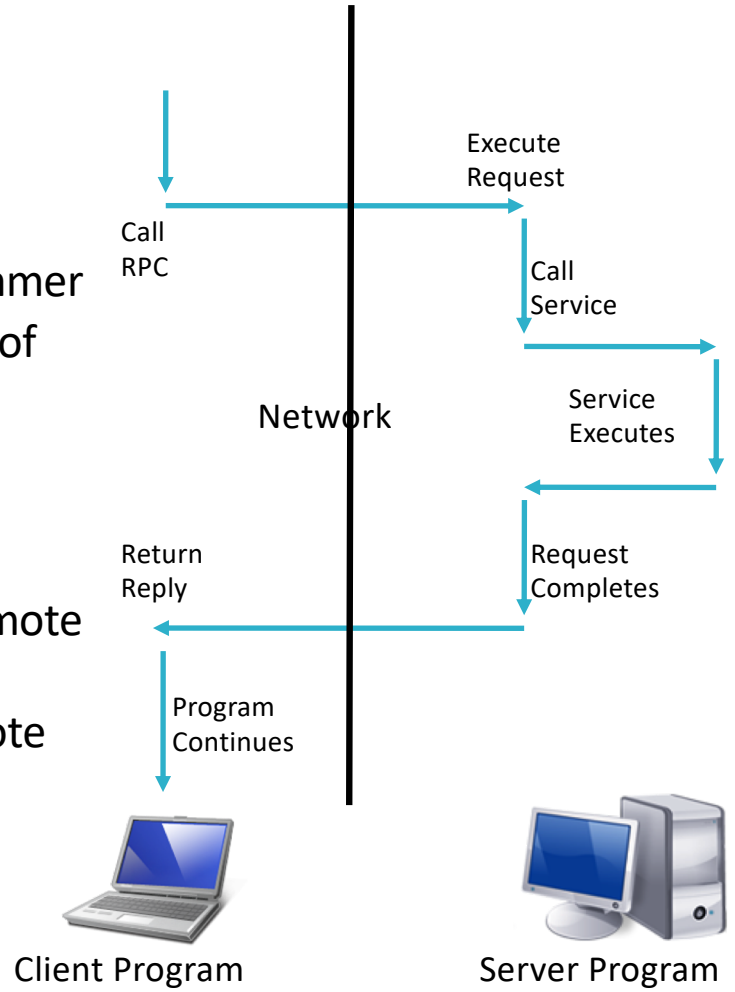
Figure 4-3 Basic RPC model

Remote Procedure Call (RPC)

- Powerful technique for constructing distributed, client-server based applications
  - Extends notion of conventional local procedure calls
  - Allows programmers of distributed applications to avoid network interface details
- Called procedure need not exist in the same address space as calling procedure
  - Two processes may be on the same system
  - They may be on different systems connected by network
- Key: transport independence
  - Isolates application from communications layer
  - Allows application to use a variety of transports

How Does It Work

- It's all about Abstractions!
  - Hide the network from the programmer
  - Clustered computing with the ease of standalone applications
- Analogous to a function call
- When an RPC is made
  - Calling arguments passed to the remote procedure
  - Caller waits for response from remote procedure
  - Blocks



The RPC Model

- RPC model supports transparent distribution of the application logic

- Remote procedures are (almost) indistinguishable from local procedures
  - *Client program invokes a function and blocks*
  - *Logic flow of execution transferred to server*
  - *Server executes the procedure and returns results to client*
  - *Client resumes execution*

Differences from Local Procedure Invocation

- Delay is substantially larger (orders of magnitude)
- Not possible to pass pointers to data structures
  - client and server operate in different address spaces
- Process environment is different (open files, open sockets, etc)
- Failure mode is different

RPC Implementation

- Version 2 standardized in RFC 1831
- Usually based on UDP (may use TCP)
- Use XDR to represent data
- Exported calls are identified by three numbers (program, version, procedure)

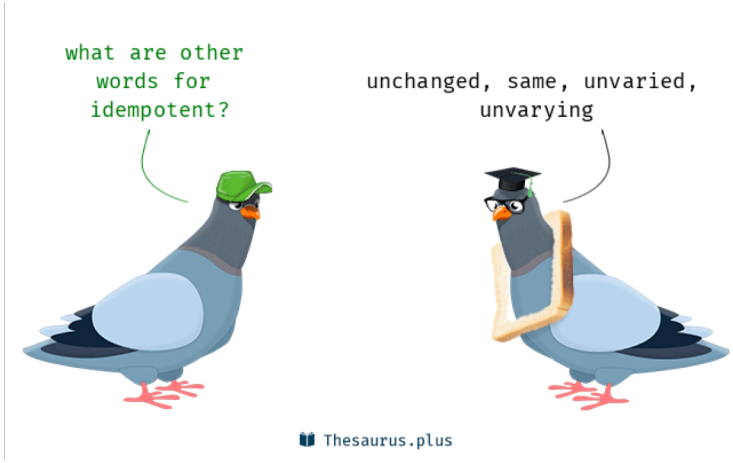
RPC Semantics

- Idempotent

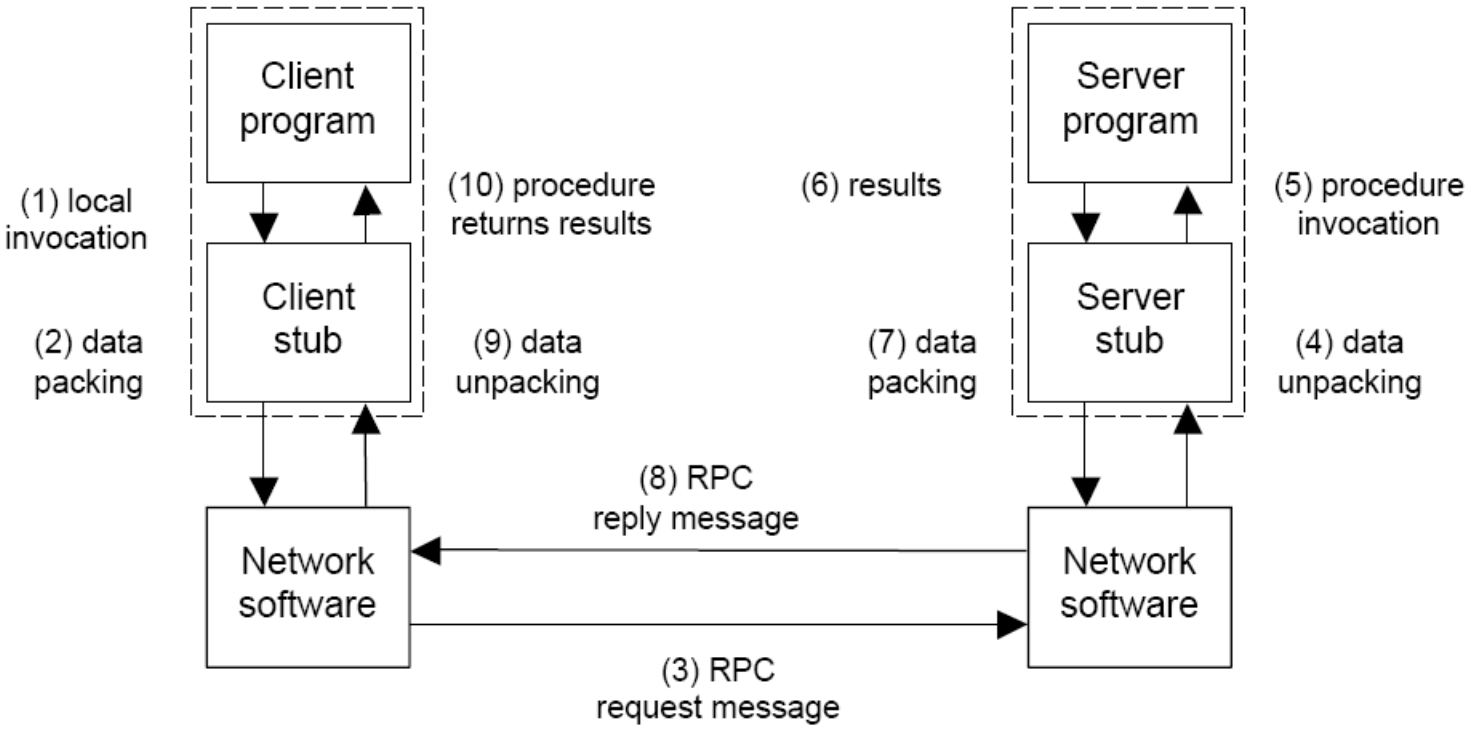
- At Least Once/Zero Or More Semantics
  - *RPC calls may be lost or duplicated (when using UDP)*
    - *If a reply is returned the caller knows that a procedure was called at least once*
    - *If no reply is received the client has to assume that the call was executed zero or more times*

Type of Procedures

- Idempotent
  - *Can be repeated more than one time without changing the results or the state of the system*
    - *e.g., read the first 5 bytes of a file*
- Non-idempotent
  - *Cannot be repeated safely*
  - *e.g., write 5 bytes to a file*



A Closer Look at RPC



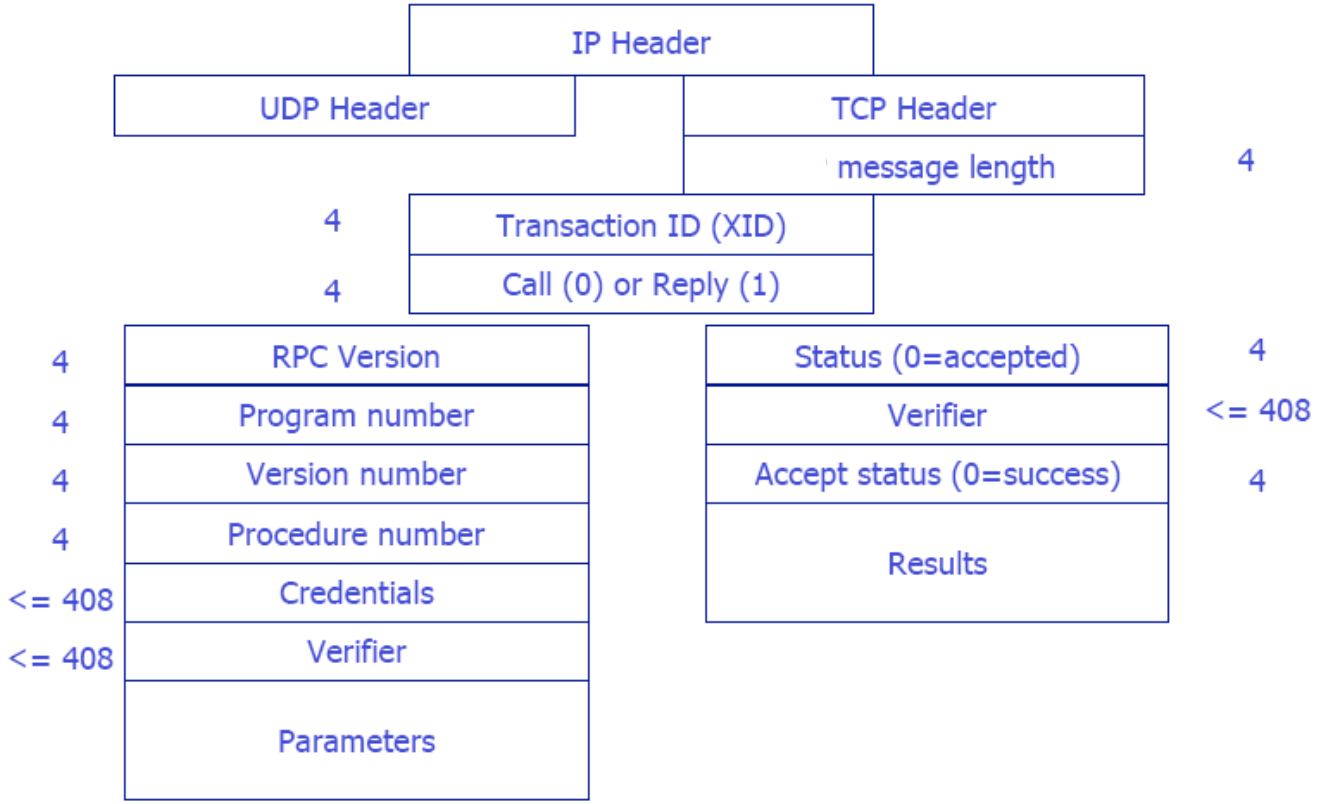
RPC Request

- Transaction ID
  - *Used to match requests and replies*
- Program, version, procedure IDs
  - *Specify the function to be invoked*
- Credentials
  - *Identify the client (e.g., user and group ID)*
- Verifier
  - *Used with Secure RPC*
- Parameters
  - *Encoded in XDR*

RPC Reply

- Transaction ID
  - *Matches the request ID*
- Status
  - *Tells if the request was accepted or not (e.g., because the server could not authenticate the client)*
- Verifier
  - *Used with Secure RPC to identify the server*
- Accept status
  - *Non-zero if invalid version or procedure were used*
- Results
  - *Encoded in XDR*

RPC Messages



Not Quite Client-Server

- Similar in operation
  - *One or more "clients" requesting services*
  - *One "server" responding*
- Problem
  - *Could have arbitrary number of services*
  - *RPC can be used for anything*
    - *Inter-process communication across two hosts*
  - *Question: how do you locate each service?*
    - *Cannot assume out-of-band knowledge of port numbers*

The Portmapper

- RPC servers do not “live” at well-known ports

- OS assigns random ports to servers, which in turn register themselves with the portmapper (well-known port 111)
  *layer of indirection!!
  - ***`PMAPPROC_SET`***
  - ***`PMAPPROC_UNSET`***

- RPC clients ask portmapper the address of the desired server, then contact server directly
  - ***`PMAPPROC_GETPORT`*** *takes program, version, and protocol as parameters*
  - ***`PMAPPROC_DUMP`*** *requests all the information*

RPC Info

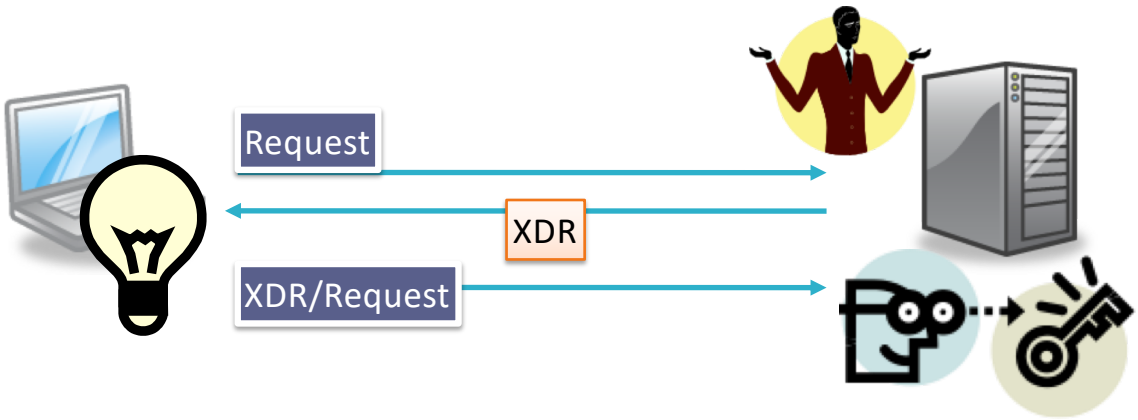
```
thor Wed Oct 25(11:31pm)[~]:-> rpcinfo -p linux.cs.uchicago.edu
   program vers proto   port
    100000    4   tcp    111  rpcbind
    100000    3   tcp    111  rpcbind
    100000    2   tcp    111  rpcbind
    100000    4   udp    111  rpcbind
    100000    3   udp    111  rpcbind
    100000    2   udp    111  rpcbind
thor Wed Oct 25(11:31pm)[~]:->
```

How do you pass a pointer to a data structure to your server?

eXternal Data Representation (XDR)

- XDR is a standard for the description and encoding of data
  - *A language for specifying the format of data in a platform independent way*
- Useful for transferring data between different computer architectures
  - *Think "serialization" in Java-speak*
  - Take your data structure, serialize it behind the scenes, send it over the network, and reassemble it properly on the other end
- Used for encoding parameters in RPC messages
- Defined in RFC 1832

Example Scenario



XDR Types

- Byte: 8 bits
- All types represented as multiples of 4-byte blocks
- Padding may be necessary
- We'll go through some basic types here

RPC specification file (. x)	C header file (. h)
const name = value;	#define name value
typedef declaration ;	typedef declaration ;
char var; short var; int var; long var; hyper var;	char var; short var; int var; long var; longlong_t var;
unsigned char var; unsigned short var; unsigned int var; unsigned long var; unsigned hyper var;	u-char var; u-short var; u-int var; u-long var; u-long long_t var ;
float var; double var; quadruple var ;	float var; double var; quadruple var;
boo1 var;	bool_t var;
enum var { name = const, . . . };	enum var { name = const, . . . } ; typedef enum var var;
opaque var[n] ;	char var[n] ;
opaque var<m>;	struct { u-int var_len; char *var_val ; } var;
string var<m>;	char *var;
datatype var [n];	datatype var[n];
datatype var<m>;	struct { u-int var_len; datatype *var_val ; } var;
struct var { members . . . };	struct var { members . . . } ; typedef struct var var;

union var switch (int disc) {
case discmlueA : armdeclA ;
case discvalueB : armdeclB ;
. . .
default : defaultdecl;
} ;
	struct var {
	int disc;
	union {
	armdeclA ;
	armdeclB ;
	. . .
	defaultdecl ;
	}var_u;
	};
	typedef s truct var var;

datatype *name;	datatype *name;

https://www.ibm.com/support/knowledgecenter/ssw_aix_72/com.ibm.aix.progcomc/xdr_datatypes.htm

Detailed info on XDR data types

Integers

- An XDR signed integer is a 32-bit datum that encodes an integer in the range [-2147483648, 2147483647]
  - *Integer represented in two's complement notation*
  - ***`int identifier;`***
- An XDR unsigned integer is a 32-bit datum that encodes a nonnegative integer in the range [0, 4294967295]
  - ***`unsigned int identifier;`***

Enumerations

- Enumerations have the same representation as signed integers
  - *Handy for describing subsets of the integers*

- Enumerated data is declared as follows:
  - ***`enum { name-identifier = constant, ... } identifier;`***

- For example, the three colors red, yellow, and blue could be described by an enumerated type
  - ***`enum { RED = 2, YELLOW = 3, BLUE = 5 } colors;`***

Bools and Floats

- The definition of a Boolean is equivalent to:
  - ***`enum { FALSE = 0, TRUE = 1 } identifier;`***
  - *It's declared as:* ***`bool identifier;`***

- Float point data (32 bits) is represented in the IEEE standard for normalized single-precision floating-point numbers (S-EF)
  - *S: The sign of the number (1 bit)*
  - *E: The exponent of the number, base 2 (8 bits)*
  - *F: The fractional part of the number's mantissa, base 2 (23 bits)*
  - $(-1)^S * 2^{E-127} * 1.F$

- It is declared as follows: **`float identifier;`**

Strings

- A string is composed of n ASCII bytes (numbered 0 through n-1)
    - *Encoded as unsigned integer n (4 bytes), followed by the n bytes of the string. If n not a multiple of four, then the n bytes are followed by (0 to 3) residual zero bytes*

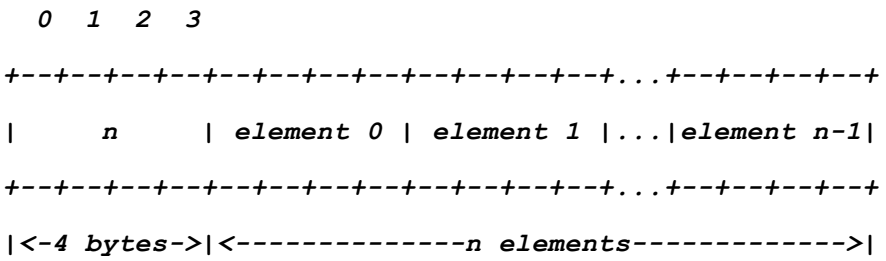
- Counted byte strings are declared as follows:
    - ***`string object<m>;`***

- Constant m denotes upper bound of the number of bytes that a string may contain. If not specified, m is assumed to be $(2^{32}) - 1$

Arrays

- Fixed-length arrays
  - *type-name identifier[n];*
  - *n elements are homogeneous (of the same type)*
  - *Elements do not need to be all of the same size (multiple of 4 bytes, though)*

- Variable-length (counted) arrays
  - *Array is encoded as the element count n followed by the encoding of each of the array's elements*
  - *type-name identifier<m>;*



Struct

- Structures are declared as follows:

```
struct name {
     component-declaration-A;
     component-declaration-B;
     ...
};
```

- Components of the structure are encoded in the order of their declaration in the structure

Const and Typedef

- Const and Typedef are used to name particular constant or types
- Const
  - *const name-identifier = n;*
  - *Example:* ***`const MAXLENGTH = 1024;`***
- Typedef
  - *typedef declaration;*
  - *Example:* ***`typedef string name<MAXLENGTH>;`***

Opaque Data

- Sometimes it is necessary to represent data with no structure

- Fixed-length opaque data
  - ***opaque identifier[n];***

- Variable-length opaque data
  - ***opaque identifier<m>;***

Other

- Discriminated union
  - ```
    union switch (discriminant-declaration) {
    case discriminant-value-A:
           arm-declaration-A;

    case discriminant-value-B:
           arm-declaration-B;

    ...

    default: default-declaration;

    } identifier;
    ```

- Void: **`void;`** there is no data here

- Optional data: type-name *identifier;
    ```

Implicit Types

- XDR specifies how data object is encoded
  - *For the set of basic types described*
  - *But encoding does not contain information on types, just the data*

- Implication
  - *Client and server must agree on exact format of messages to be exchanged*
  - *Program must have exact formatting and types of a message to decode it*

Example

```
const MAXUSERNAME = 32;   /* max length of a user name */
const MAXFILELEN = 65535; /* max length of a file */
const MAXNAMELEN = 255;   /* max length of a file name */
/* Types of files */
enum filekind {
  TEXT = 0, /* ascii data */
  DATA = 1, /* raw data */
  EXEC = 2 /* executable */
};
```

Example

```
/* File information, per kind of file */
union filetype switch (filekind kind) {
 case TEXT:
   void;                              /* no extra information */
 case DATA:
   string creator<MAXNAMELEN>;        /* data creator */
 case EXEC:
    string interpreter<MAXNAMELEN>; /* program interpreter */
};
```

Example

```
/* A complete file */
struct file {
   string filename<MAXNAMELEN>; /* name of file    */
   filetype type;               /* info about file */
   string owner<MAXUSERNAME>;   /* owner of file   */
   opaque data<MAXFILELEN>;     /* file data       */
};
```

```
const MAXUSERNAME = 32;     /*  max length of a user name  */
const MAXFILELEN = 65535;   /*  max length of a file       */
const MAXNAMELEN = 255;     /*  max length of a file name  */
/*
 * Types of files:
 */
enum filekind {
    TEXT = 0,       /*  ascii data  */
    DATA = 1,       /*  raw data    */
    EXEC = 2        /*  executable  */
};
/*
 * File information, per kind of file:
 */
union filetype switch (filekind kind) {
    case TEXT:
        void;               /*  no extra information  */
    case DATA:
        string creator<MAXNAMELEN>;   /*  data creator  */
    case EXEC:
        string interpretor<MAXNAMELEN>; /* program interpretor */
};
/*
 * A complete file:
 */
struct file {
    string filename<MAXNAMELEN>; /*  name of file  */
    filetype type;               /*  info about file  */
    string owner<MAXUSERNAME>;   /*  owner of file    */
    opaque data<MAXFILELEN>;     /*  file data        */
};
```

Encoding

- Suppose now that there is a user named "john“
- He wants to store his lisp program named "sillyprog“
- The file contains just the data "(quit)"

Encoding

```
OFFSET  HEX BYTES   ASCII   COMMENTS
------  ---------   -----   --------
0       00 00 00 09 ....    -- length of filename = 9
4       73 69 6c 6c sill    -- filename characters
8       79 70 72 6f ypro    -- ... and more characters
12      67 00 00 00 g...    -- ... and 3 zero-bytes of fill
16      00 00 00 02 ....    -- filekind is EXEC = 2
20      00 00 00 04 ....    -- length of interpreter = 4
24      6c 69 73 70 lisp    -- interpreter characters
28      00 00 00 04 ....    -- length of owner = 4
32      6a 6f 68 6e john    -- owner characters
36      00 00 00 06 ....    -- length of file data = 6
40      28 71 75 69 (qui    -- file data bytes ...
44      74 29 00 00 t)..    -- ... and 2 zero-bytes of fill
```

Creating RPC Programs

- RPC mechanisms can be used in different ways/levels
  - *Encoding of both the parameters and request message can be done manually*
  - *A procedure can be made to invoke the remote function, providing function pointer for marshalling/unmarshalling of the parameters*
  - *Can use the RPC library and the rpcgen tool (RPC compiler)*

rpcgen

- rpcgen is a tool that takes care of
  - *Producing the stubs for the client- and server-side procedures*
  - *Producing the code that marshals/unmarshals the parameters*
  - *Producing the main server loop*

- Analogous to Java counterparts
  - *Stub-compilers*
  - *rmiserver*

Development Process

- Write the procedure to be invoked remotely
- Write the client that invokes the procedures
- Write a specification of the procedures in the RPC interface definition language (IDL)
- Generate the stubs for client and server sides
- Write client and server interfaces that map the program's calling convention to RPC's calling convention

Here's What It Looks Like