Inter-machine communication

- **Datagram sockets: Unreliable message delivery**
  - User Datagram Protocol (UDP/IP)
  - Send atomic messages, which may be reordered or lost
- **Stream sockets: Bi-directional pipes**
  - Transmission Control Protocol (TCP/IP)
  - Bytes written on one end read on the other

Socket naming

- **Every Internet host has a unique 32-bit *IP address***
  - Often written in "dotted-quad" notation: 204.168.181.201
  - DNS protocol maps names (www.nyu.edu) to IP addresses
  - Network routes packets based on IP address
- **16-bit *port number* demultiplexes TCP traffic**
  - Well-known services "listen" on standard ports: finger—79, HTTP—80, mail—25, ssh—22
  - Clients connect from arbitrary ports to well known ports
  - A connection consists of five components: Protocol (TCP), local IP, local port, remote IP, remote port

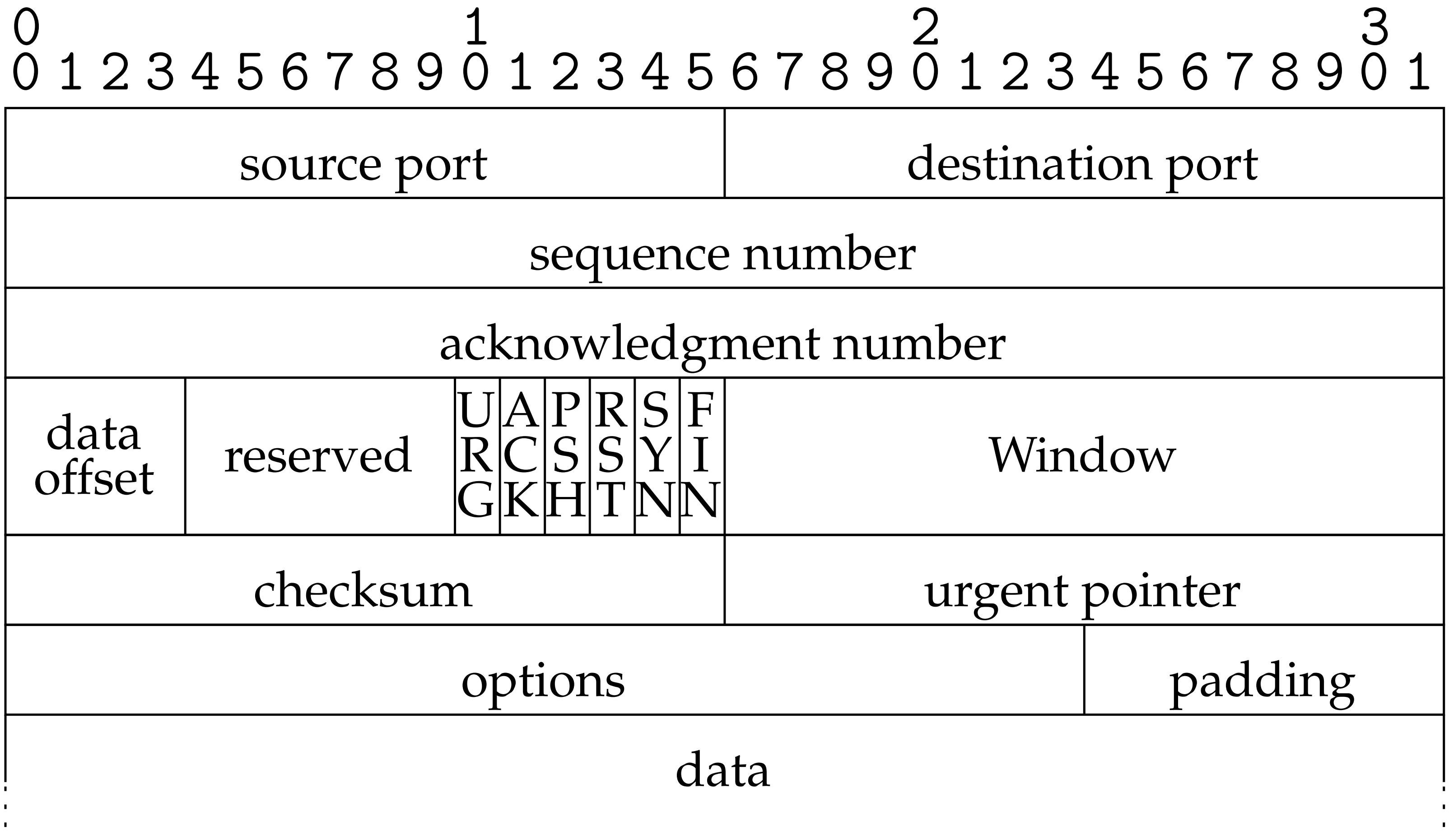
IP header

<table>
<tr><td>0</td><td></td><td></td><td></td><td></td><td></td><td></td><td></td><td></td><td></td><td>1</td><td></td><td></td><td></td><td></td><td></td><td></td><td></td><td></td><td></td><td>2</td><td></td><td></td><td></td><td></td><td></td><td></td><td></td><td></td><td></td><td>3</td><td></td></tr>
<tr><td>0</td><td>1</td><td>2</td><td>3</td><td>4</td><td>5</td><td>6</td><td>7</td><td>8</td><td>9</td><td>0</td><td>1</td><td>2</td><td>3</td><td>4</td><td>5</td><td>6</td><td>7</td><td>8</td><td>9</td><td>0</td><td>1</td><td>2</td><td>3</td><td>4</td><td>5</td><td>6</td><td>7</td><td>8</td><td>9</td><td>0</td><td>1</td></tr>
<tr><td colspan="4">vers</td><td colspan="4">hdr len</td><td colspan="8">TOS</td><td colspan="16">Total Length</td></tr>
<tr><td colspan="16">Identification</td><td>0</td><td>D
F</td><td>M
F</td><td colspan="13">Fragment offset</td></tr>
<tr><td colspan="8">TTL</td><td colspan="8">Protocol</td><td colspan="16">hdr checksum</td></tr>
<tr><td colspan="32">Source IP address</td></tr>
<tr><td colspan="32">Destination IP address</td></tr>
<tr><td colspan="24">Options</td><td colspan="8">Padding</td></tr>
</table>

IP header details

- **Routed to destination address specified**
- **TTL (time to live) decremented at each hop (avoids loops)**
- **Fragmentation used for large packets**
  - Fragmented in network if links crossed with smaller MTU
  - MF bit means more fragments for this IP packet
  - DF bit says “don’t fragment” (returns error to sender)

TCP header


0 1 2 3
0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1
source port
destination port
sequence number
acknowledgment number
data offset
reserved
URG
ACK
PSH
RST
SYN
FIN
Window
checksum
urgent pointer
options
padding
data

TCP fields

- **Ports**
- **Seq no. – segment position in byte stream**
- **Ack no. – seq no. sender expects to receive next**
- **Data offset – # of 4-byte header & option words**
- **Window – willing to receive (flow control)**
- **Checksum**
- **Urgent pointer**

TCP Flags

- **URG – urgent data present**
- **ACK – ack no. valid (all but first segment)**
- **PSH – push data up to application immediately**
- **RST – reset connection**
- **SYN – "synchronize" establishes connection**
- **FIN – close connection**

A TCP Connection (no data)

```
orchard.48150 > essex.discard:
        S 1871560457:1871560457(0) win 16384
essex.discard > orchard.48150:
        S 3249357518:3249357518(0) ack 1871560458 win 17376
orchard.48150 > essex.discard: . ack 1 win 17376
orchard.48150 > essex.discard: F 1:1(0) ack 1 win 17376
essex.discard > orchard.48150: . ack 2 win 17376
essex.discard > orchard.48150: F 1:1(0) ack 2 win 17376
orchard.48150 > essex.discard: . ack 2 win 17375
```

Connection establishment

- **Three-way handshake:**
  - $C \rightarrow S$: SYN, seq S_C
  - $S \rightarrow C$: SYN, seq S_S, ack $S_C + 1$
  - $C \rightarrow S$: ack $S_S + 1$
- **If no program listening: server sends RST**
- **If server backlog exceeded: ignore SYN**
- **If no SYN-ACK received: retry, timeout**

Connection termination

- **FIN bit says no more data to send**
  - Caused by close or shutdown on sending end
  - Both sides must send FIN to close a connection
- **Typical close:**
  - $A \rightarrow B$: FIN, seq S_A, ack S_B
  - $B \rightarrow A$: ack $S_A + 1$
  - $B \rightarrow A$: FIN, seq S_B, ack $S_A + 1$
  - $A \rightarrow B$: ack $S_B + 1$
- **Can also have simultaneous close**
- **After last message, can A and B forget about closed socket?**

TIME_WAIT

- **Problems with closed socket**
  - What if final ack is lost in the network?
  - What if the same port pair is immediately reused for a new connection? (Old packets might still be floating around.)
- **Solution: “active” closer goes into TIME_WAIT**
  - Active close is sending FIN before receiving one
  - After receiving ACK and FIN, keep socket around for 2MSL (twice the “maximum segment lifetime”)

Sending data

- **Data sent in MSS-sized segments**
  - Chosen to avoid fragmentation (e.g., 1460 on ethernet LAN)
  - Write of 8K might use 6 segments—PSH set on last one
  - PSH avoids unnecessary context switches on receiver
- **Sender's OS can delay sends to get full segments**
  - Nagle algorithm: Only one unacknowledged short segment
  - TCP_NODELAY option avoids this behavior
- **Segments may arrive out of order**
  - Sequence number used to reassemble in order
- **Window achieves flow control**
  - If window 0 and sender's buffer full, write will block or return EAGAIN

A TCP connection (3 byte echo)

```
orchard.38497 > essex.echo:
        S 1968414760:1968414760(0) win 16384
essex.echo > orchard.38497:
        S 3349542637:3349542637(0) ack 1968414761 win 17376
orchard.38497 > essex.echo: . ack 1 win 17376
orchard.38497 > essex.echo: P 1:4(3) ack 1 win 17376
essex.echo > orchard.38497: . ack 4 win 17376
essex.echo > orchard.38497: P 1:4(3) ack 4 win 17376
orchard.38497 > essex.echo: . ack 4 win 17376
orchard.38497 > essex.echo: F 4:4(0) ack 4 win 17376
essex.echo > orchard.38497: . ack 5 win 17376
essex.echo > orchard.38497: F 4:4(0) ack 5 win 17376
orchard.38497 > essex.echo: . ack 5 win 17375
```

Retransmission

- TCP dynamically estimates round trip time
- If segment goes unacknowledged, must retransmit
- Use exponential backoff (in case loss from congestion)
- After $\sim$10 minutes, give up and reset connection

Bro: Detecting network intruders

- **Many security holes exploited over the network**
  - Buffer overruns in servers
  - Servers with bad implementations
    ("login -froot", telnet w. LD_LIBRARY_PATH)
- **Goal: Detect people exploiting such bugs**
- **Detect activities performed by people who've penetrated server**
  - Setting up IRC bot
  - Running particular commands, etc.

Bro model

- **Attach machine running Bro to "DMZ"**
  - Demilitarized zone – area betw. firewall & outside world
- **Sniff all packets in and out of the network**
- **Process packets to identify possible intruders**
  - Secret, per-network rules identify possible attacks
  - Is it a good idea to keep rules secret?
- **React to any threats**
  - Alert administrators of problems in real time
  - Switch on logging to enable later analysis of potential attack
  - Take action against attackers – E.g., filter all packets from host that seems to be attacking

Goals of system

- **Keep up with high-speed network**
  - No packet drops
- **Real-time notification**
- **Separate mechanism from policy**
  - Avoid easy mistakes in policy specification
  - So different sites can specify “secret” policies easily
- **Extensibility**
- **Resilience to attack**

Challenges

- **Have to keep up with fast packet rate**
- **System has to be easy to program**
  - Every site needs different, secret rules
  - Don’t want system administrators making mistakes
- **Overload attacks**
- **Crash attacks**
- **Subterfuge attacks**

Bro architecture

- **Layered architecture:**
  - bpf/libpcap, Event Engine, Policy Script Interpreter
- **Lowest level bpf filter in kernel**
  - Match interesting ports or SYN/FIN/RST packets
  - Match IP fragments
  - Other packets do not get forwarded to higher levels
- **Event engine, written in C++**
  - Knows how to parse particular network protocols
  - Has per-protocol notion of events
- **Policy Script Interpreter**
  - Bro language designed to avoid easy errors

Overload and Crash attacks

- **Overload goal: prevent monitor from keeping up w. data stream**
  - Leave exact thresholds secret
  - Shed load (e.g., HTTP packets)
- **Crash goal: put monitor out of commission**
  - E.g., run it out of space (too much state)
  - Watchdog timer kills & restarts stuck monitor
  - Also starts tcpdump log, so same crash attack, if repeated, can be analyzed

Questions

- Why is FTP more complicated? How to handle?
- How to deal with type-ahead in telnet/rlogin connections?
- What are network scans? Port scans? How to detect these?
- How convinced are we of effectiveness?

Subterfuge attacks

- **IP fragments too small to see TCP header**
- **Retransmitted IP fragments w. different data**
- **Retransmitted TCP packets w. different data**
- **TTL/MTU monkeying can hide packets from destination**
  - Compare TCP packet to retransmitted copy
  - Assume one of two endpoints is honest (exploit ACKs)
  - Bifurcating analysis

State and checkpointing

- **Need to keep a lot of session state**
  - Open TCP connections, UDP request-response, IP fragments
  - No timers to garbage collect state
- **Checkpointing the system**
  - Start new copy of monitoring process
  - Kill old copy when new copy has come up to speed
  - Is this ideal?

Stretch break

Much read-only data improperly trusted

- **People install/upgrade software over the Internet**
  - No guarantee you are talking to the right host
  - No guarantee server has not been compromised
  - No guarantee you can trust a mirror site's owner
- **Central servers configure/upgrade machines**
  - *sup*, anonymous *rsync*, AFS read-only—all insecure
- **People base financial decisions on public data**
  - Stock quotes, financial news

Why people avoid security

- **Performance**
  - Public-key cryptography can cripple throughput (e.g. SSL)
- **Scalability and reliability**
  - Widespread replication essential for popular data
  - The more replicas, the less they can be trusted
- **Convenience**
  - Most users will skip optional verification steps
  - Often hard to understand precise security guarantees

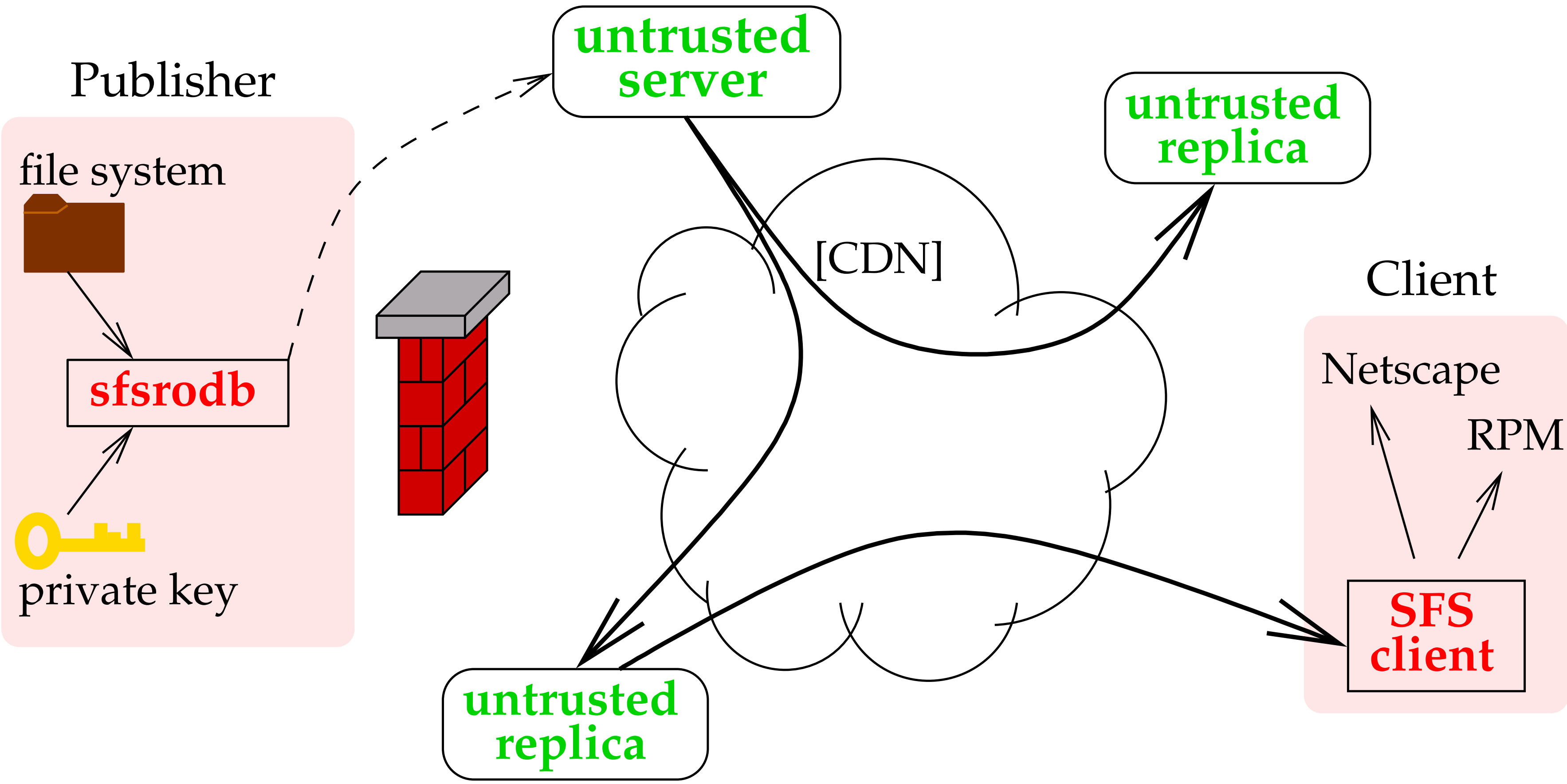
Example: PGP-sign data off-line

- **Advantages:**
  - Compromising server does not circumvent PGP security
  - Data can be replicated on untrusted servers
- **Not general purpose**
- **Most users will ignore signatures**
- **Requires continued attention of user**
  - Was file signed by authoritative key?
  - Is a signed file the latest version?
  - Does signed contents of file match file name?
  - Were two separately signed files published together?

Solution: SFS read-only file system

- **Convenience: Use the file system interface**
  - Publish any data
  - Access it from any application
- **Scalability: Separate publishing from distribution**
  - Off-line publisher produces signed database
  - On-line servers/replicas completely untrusted
- **Intrinsic security: Nothing for user to do**
  - Every file system has a public key (specified in name)
  - Client automatically verifies integrity of files

SFSRO Architecture

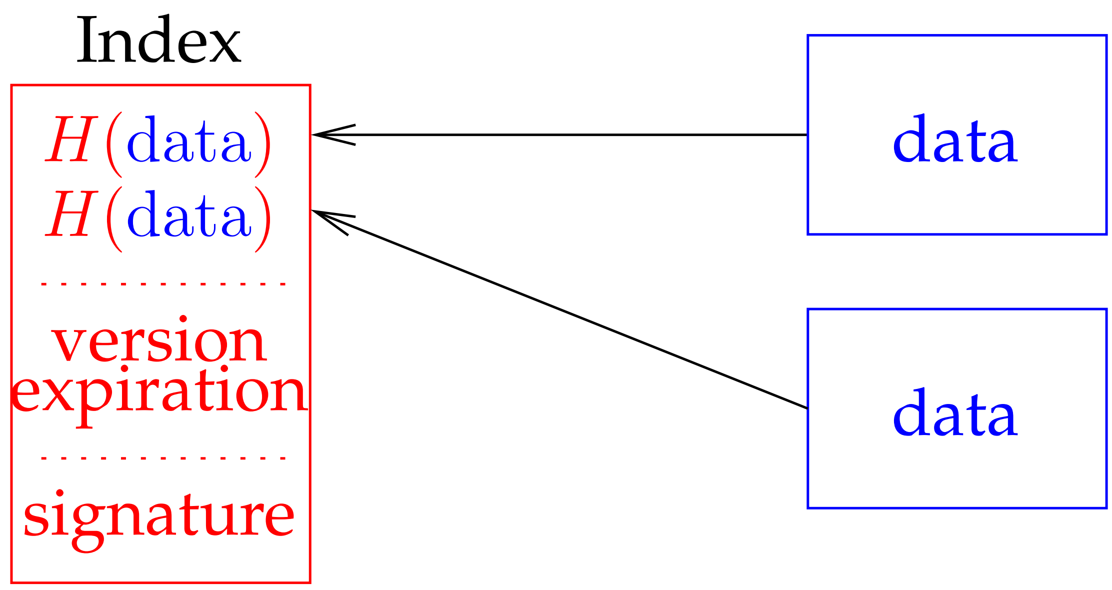


- **Publisher stores files in replicated database**
- **Clients verify files without trusting servers**

Cryptographic primitives

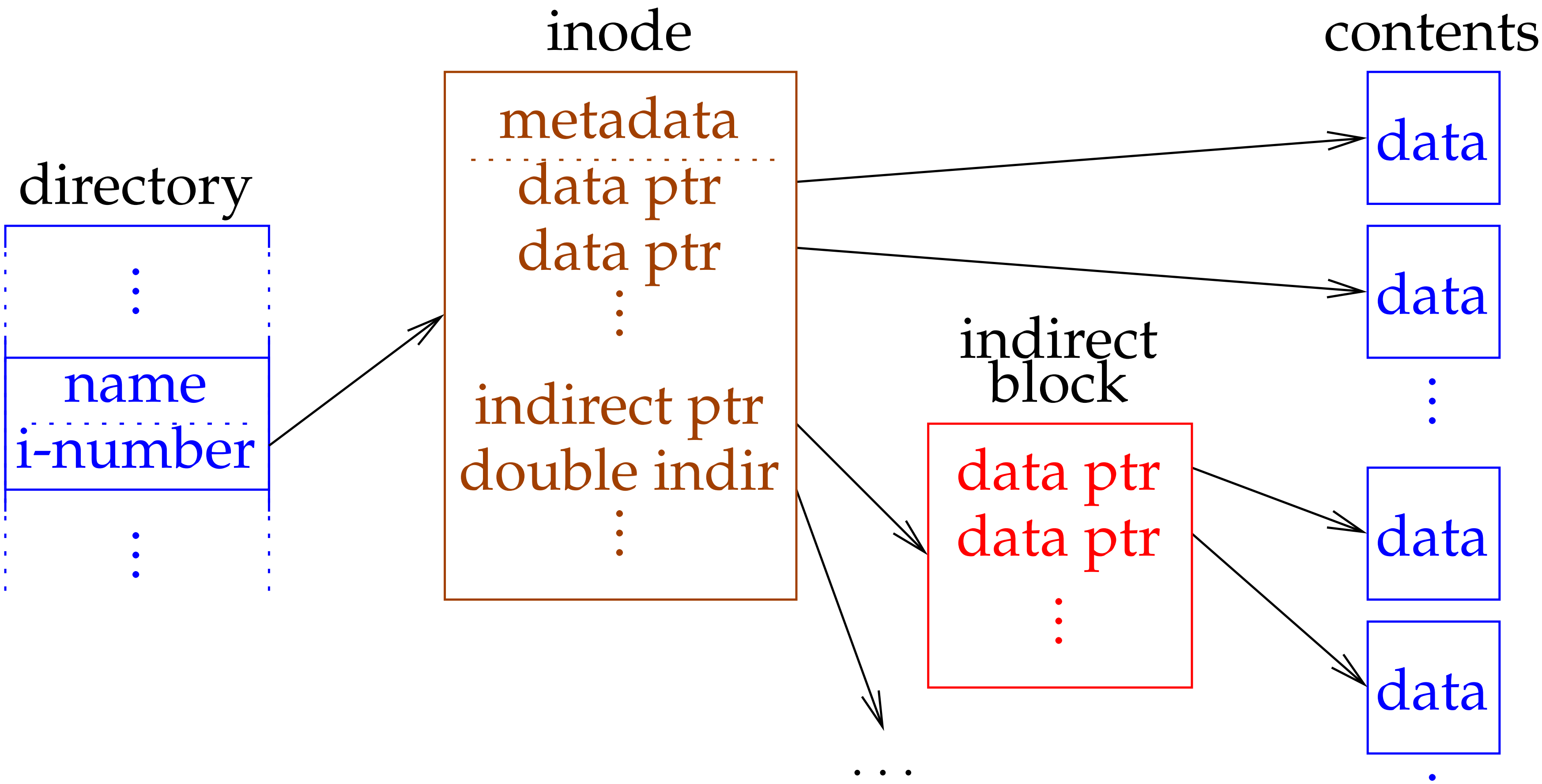
- **Digital signatures**
  - Client knows server public key in advance
  - When server signs data, client can verify integrity
  - Cost: $\sim$24 msec to sign, $\sim$80 μsec to verify
  - If server signs multiple versions, must ensure freshness
- **Collision-resistant hashes (Computationally infeasible to find** $x \neq y,\ H(x) = H(y)$**)**
  - Server hashes data securely, transmits hash to client
  - Client hashes untrusted copy, compares to trusted hash
  - Cost: 15+ MBytes/sec to hash

Example: Publishing 2 blocks of data



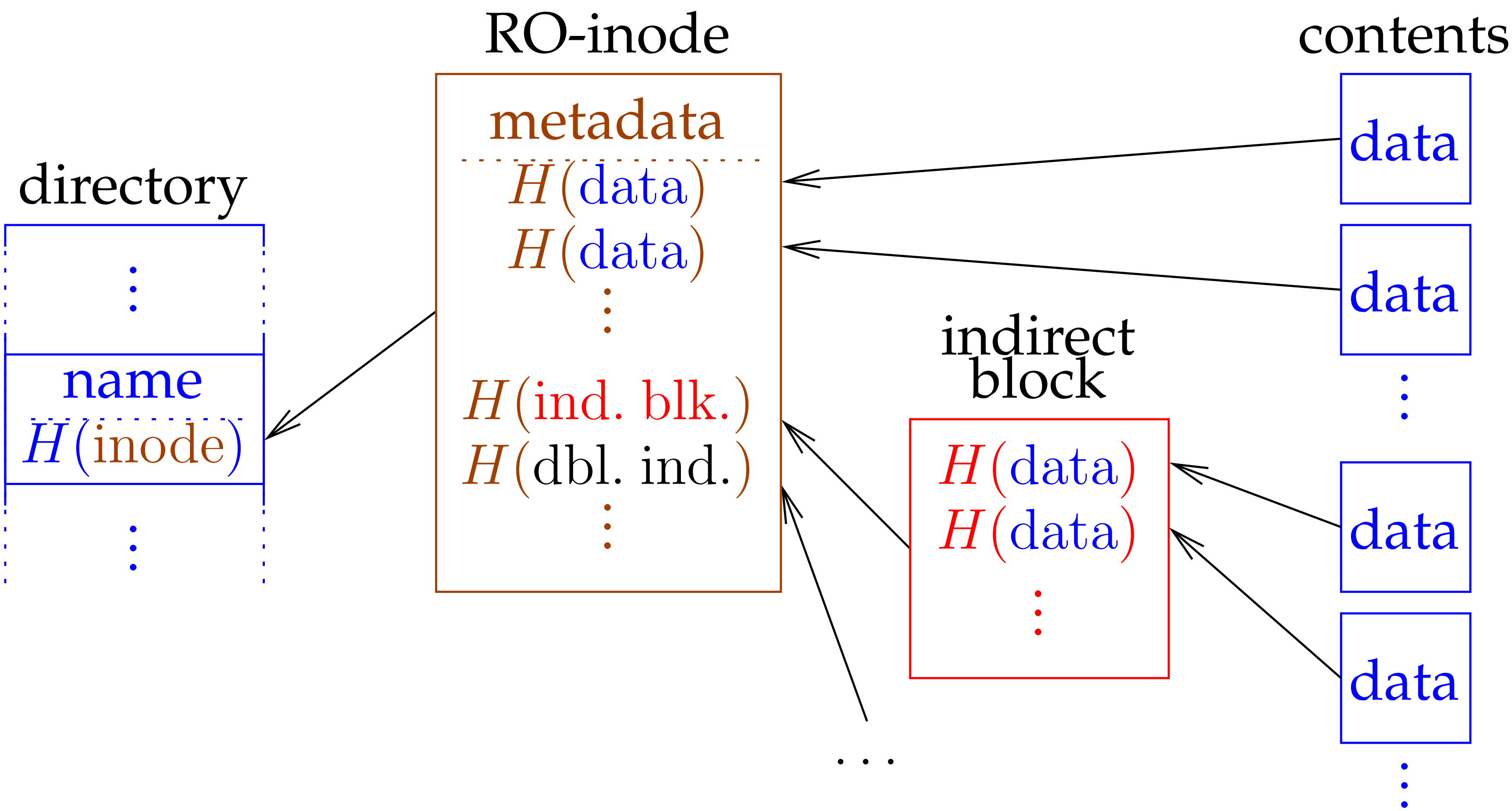
- **Digitally sign version & hashes of blocks**
  - Can verify one block without having the other
  - Two blocks must come from same version of file
- **Generalize technique to an entire file system**

Traditional FS data structures



- **In database arbitrary key can replace disk location**

Read-only data structures



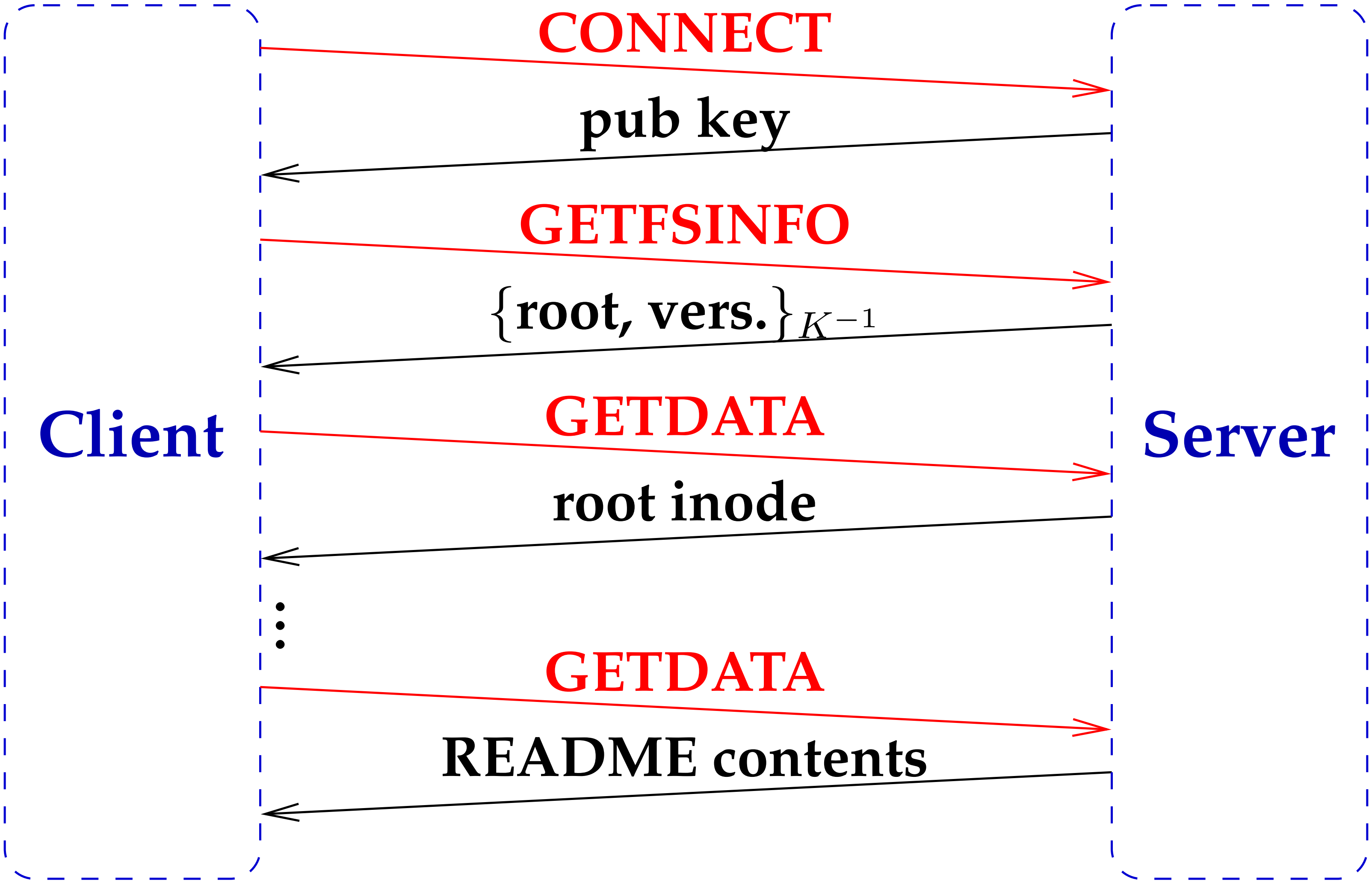
- **Index all data & metadata with cryptographic hash**

The SFSRO protocol

- **CONNECT () – Initiate SFSRO protocol**
- **GETFSINFO () – Get signed hash of root directory**
- **GETDATA (*hash*) – Get block with *hash* value**
- **All data interpreted entirely by client**
  - Server need know nothing about file system structure
  - Makes server fast and simple ($<$ 400 lines of code)

Example: File Read

`/sfs/sfs.nyu.edu:bzcc5hder7cuc86kf6qswyx6yuemnw69/README`



Incremental replication

- **Replicas need transfer only modified data**
- ***pulldb* utility incrementally updates a replica**
  - Uses SFSRO protocol to traverse file system
  - Stores all hashes/values encountered in new database
  - Avoids transferring any hashes already in old database
  - Unchanged directories automatically pruned from transfer
- **Makes short signature durations practical**

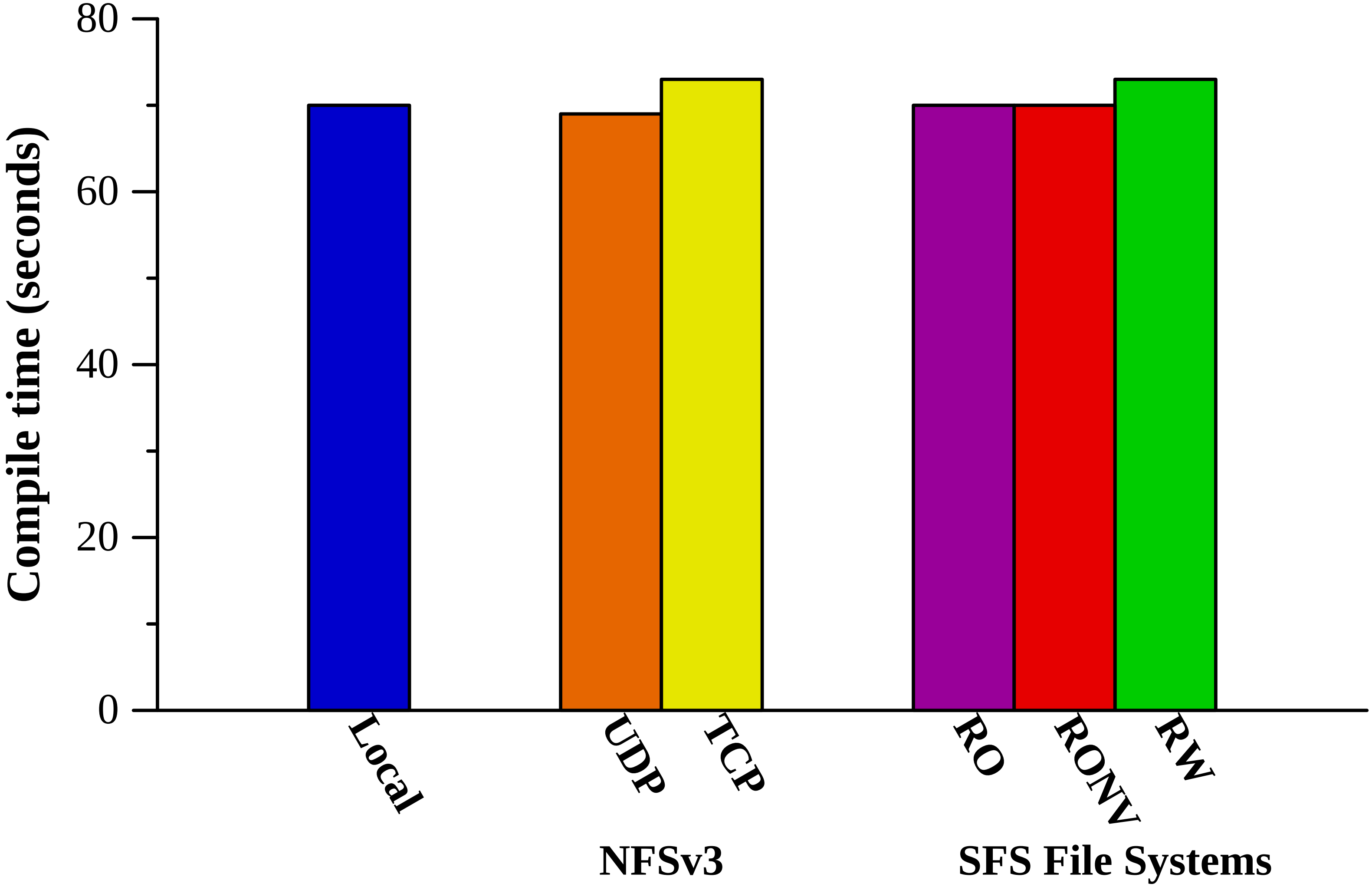
Application: RedHat distribution

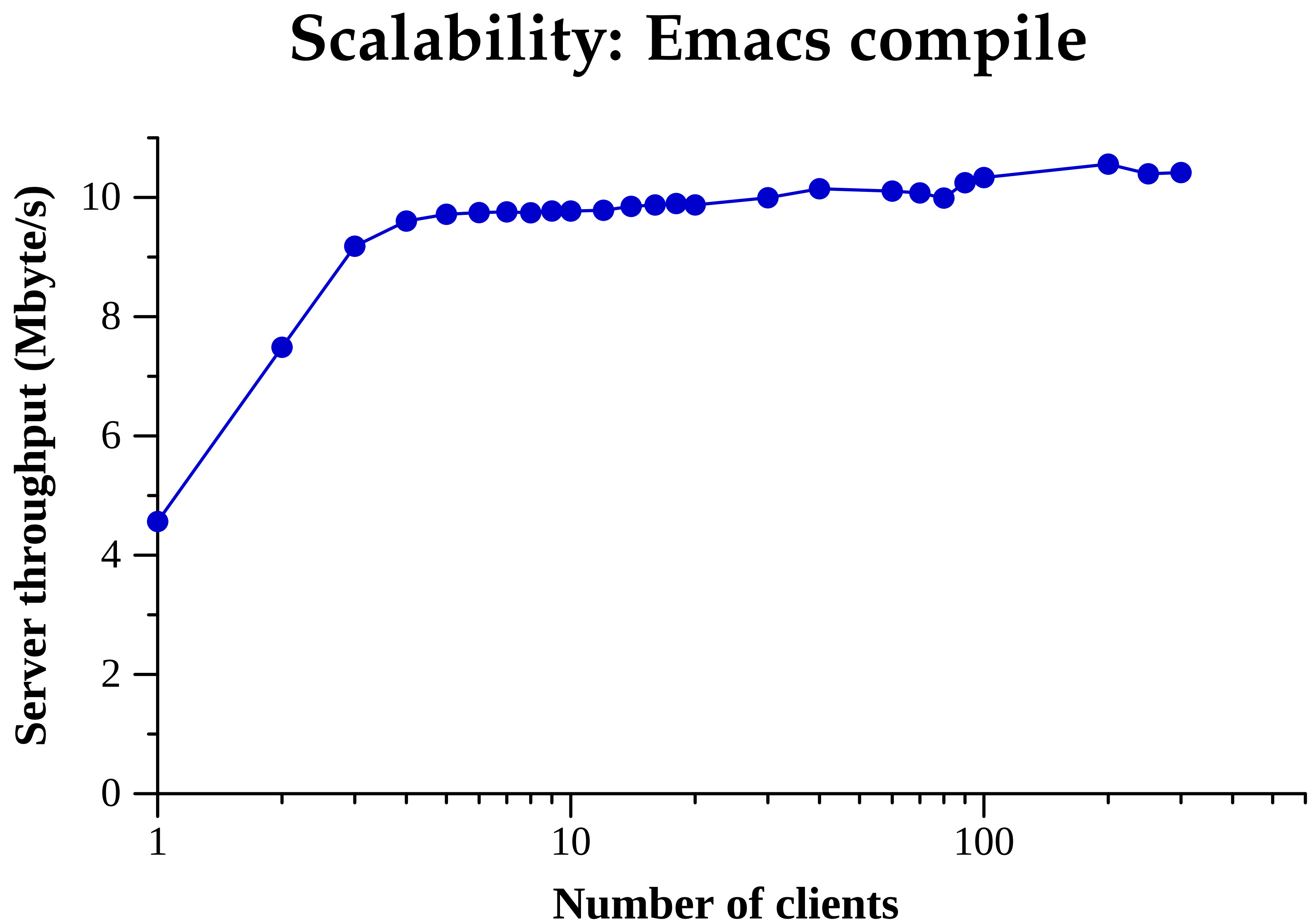
- **Publish `ftp.redhat.com` via SFSRO**
  - Push out new signature every 24 hours
- **Advantages:**
  - Volunteer mirror sites need no longer be trusted
  - Install from file system, not URL (easier to browse)
  - Secure automatic upgrade becomes a simple script
  - Can revoke/update flawed packages quickly
  - File names securely bound to contents
  - Easy for users to understand security properties

Application: Software distribution

- **Distribute open-source software via SFSRO**
  - Users can compile directly from the distribution
- **Benchmark: Compile emacs-20.6 from source code**
  - 550 MHz Pentium IIIs, 256 MBytes RAM, FreeBSD 3.3
  - Warm server cache, cold client cache

Performance: Emacs compile




Scalability: Emacs compile
Server throughput (Mbyte/s)
Number of clients
0
2
4
6
8
10
1
10
100

Application: Certificate authorities

- **SFS specifies public keys of servers in file names:**

  `/sfs/sfs.nyu.edu:bzcc5hder7cuc86kf6qswyx6yuemnw69`

- **Symbolic links hide public keys from users:**

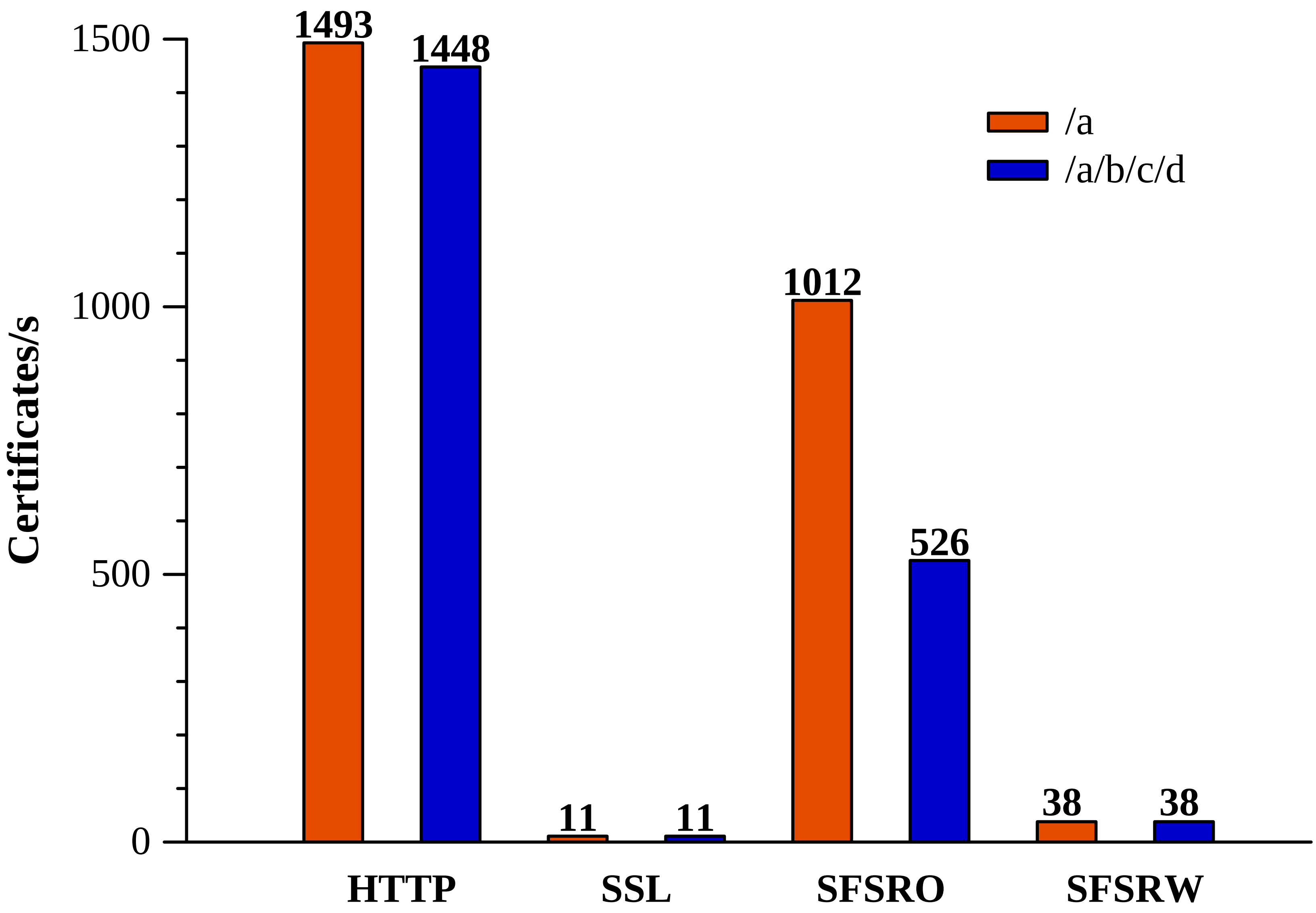
  `/verisign` → `/sfs/sfs.verisign.com:r6ui9gw···pfbz4pe`

- **SFSRO can serve name-to-key bindings:**

  `/verisign/nyu` → `/sfs/sfs.nyu.edu:bzcc5hd···uemnw69`

- **Better revocation than traditional CAs**
  - Signature can realistically expire in hours, not months
  - Cannot revert one certificate without reverting them all

Scalability: Certificate downloads



Conclusions

- **Public read-only data needs integrity guarantees.**
- **Cannot realistically sacrifice performance, scalability, or convenience to get those guarantees.**
- **SFSRO achieves integrity without sacrifice**
  - Off-line publishing has cost independent of server load
  - Dirt-simple server offloads cryptographic costs to clients
  - File system is the most convenient/universal interface