# Principles of Software Construction: Objects, Design, and Concurrency

Part 6: Concurrency and distributed systems

**Distributed systems**

Christian Kästner **Charlie Garrod**

School of Computer Science

institute for SOFTWARE RESEARCH

# Administrivia

- Homework 5b due Thursday, 11:59 p.m.
  - Finish by Friday (10 Apr) 10 a.m. if you want to be considered as a "Best Framework" for Homework 5c
    - Our evaluation considers:
      - Novelty
      - Functional correctness
      - Documentation
      - …

# Key concepts from last Thursday

# Concurrency at the language level

- Consider:

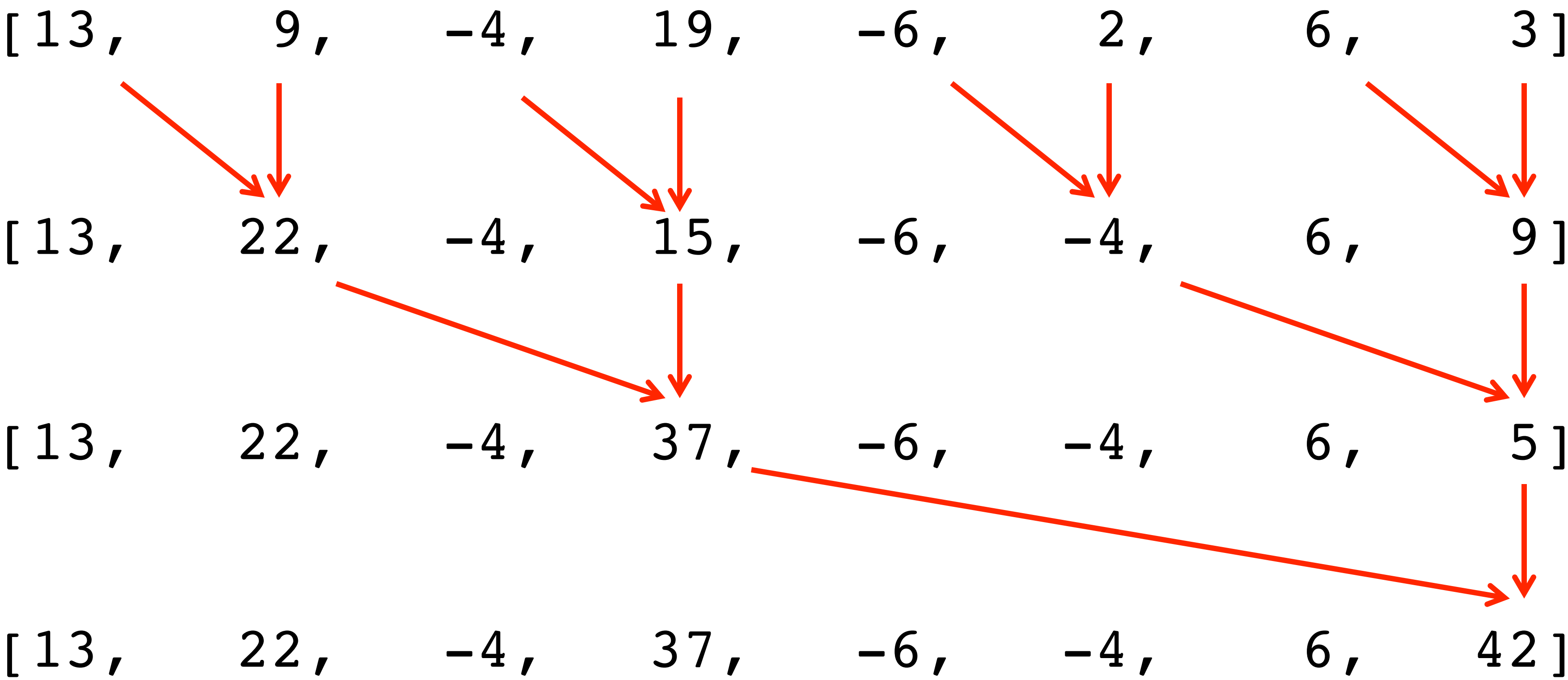
```
int sum = 0;
Iterator i = coll.iterator();
while (i.hasNext()) {
    sum += i.next();
}
```

- In python:

```
sum = 0;
for item in coll:
    sum += item
```

# Parallel prefix sums algorithm, winding

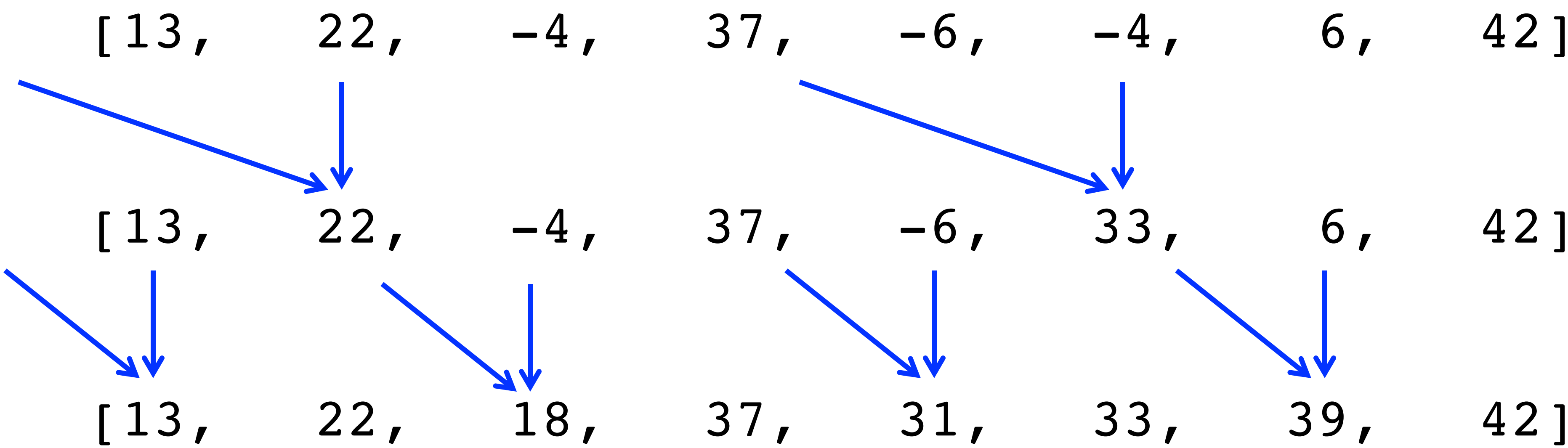
- Computes the partial sums in a more useful manner




⋮

# Parallel prefix sums algorithm, unwinding

- Now unwinds to calculate the other sums




- Recall, we started with:

[13, 9, -4, 19, -6, 2, 6, 3]

# A framework for asynchronous computation

- The `java.util.concurrent.Future<V>` interface

```
V       get();
V       get(long timeout, TimeUnit unit);
boolean isDone();
boolean cancel(boolean mayInterruptIfRunning);
boolean isCancelled();
```

- The `java.util.concurrent.ExecutorService` interface

```
Future          submit(Runnable task);
Future<V>       submit(Callable<V> task);
List<Future<V>> invokeAll(Collection<Callable<V>> tasks);
Future<V>       invokeAny(Collection<Callable<V>> tasks);
```

# Fork/Join: another common computational pattern

- In a long computation:
  - Fork a thread (or more) to do some work
  - Join the thread(s) to obtain the result of the work
- The `java.util.concurrent.ForkJoinPool` class
  - Implements `ExecutorService`
  - Executes `java.util.concurrent.ForkJoinTask<V>` or
    `java.util.concurrent.RecursiveTask<V>` or
    `java.util.concurrent.RecursiveAction`

# Parallel prefix sums algorithm

- How good is this?
  - Work: O(n)
  - Depth: O(lg n)
- See PrefixSumsSequentialImpl.java
  - n-1 additions
  - Memory access is sequential
- For PrefixSumsNonsequentialImpl.java
  - About 2n useful additions, plus extra additions for the loop indexes
  - Memory access is non-sequential
- The punchline:  Constants matter.

# Today: Distributed system design

- Java networking fundamentals
- Introduction to distributed systems
  - Motivation: reliability and scalability
  - Failure models
  - Techniques for:
    - Reliability (availability)
    - Scalability
    - Consistency

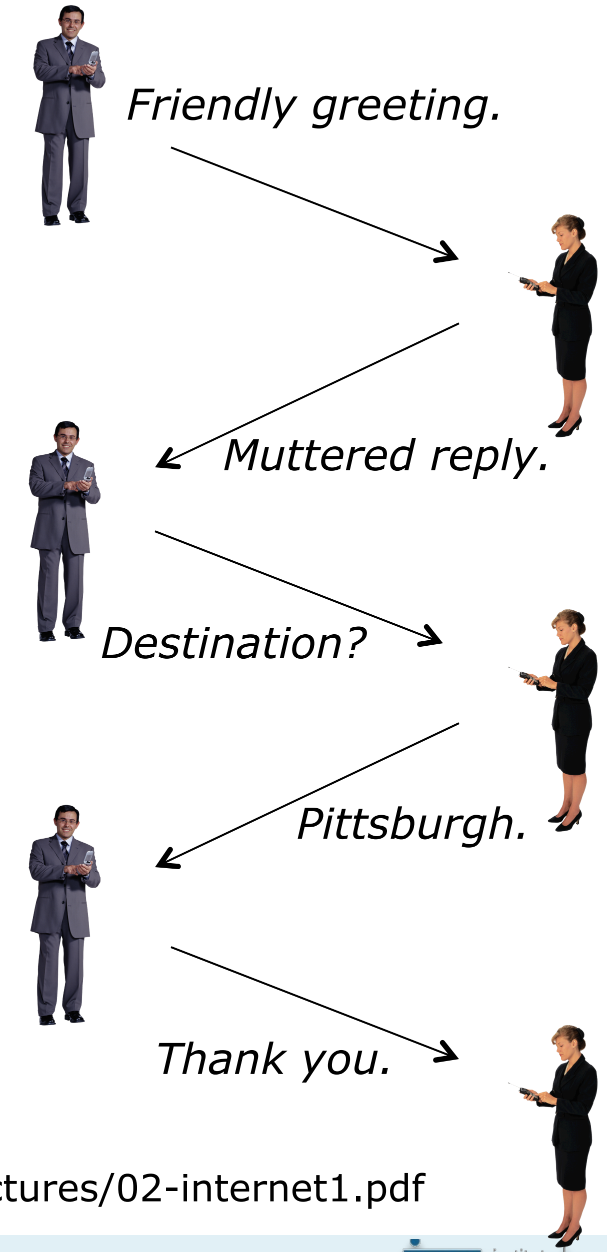
# Our destination: Distributed systems

- Multiple system components (computers) communicating via some medium (the network)
- Challenges:
  - Heterogeneity
  - Scale
  - Geography
  - Security
  - Concurrency
  - Failures

(courtesy of http://www.cs.cmu.edu/~dga/15-440/F12/lectures/02-internet1.pdf

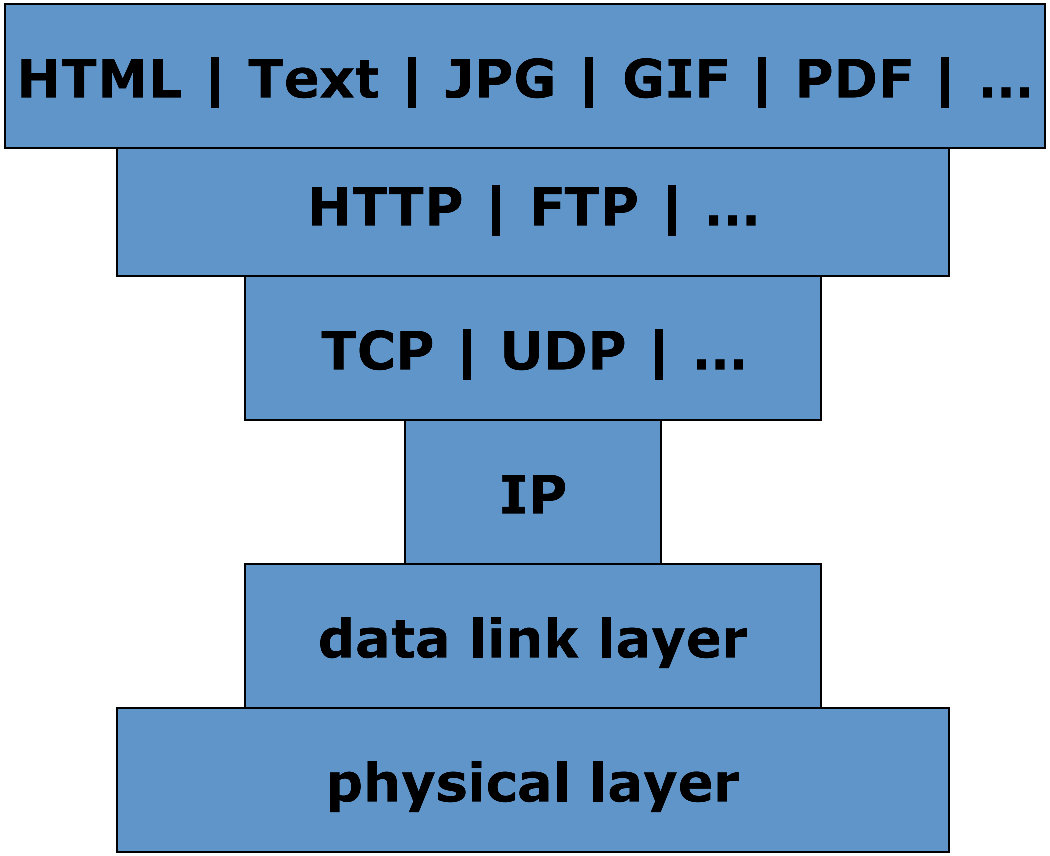
# Communication protocols

- Agreement between parties for how communication should take place




(courtesy of http://www.cs.cmu.edu/~dga/15-440/F12/lectures/02-internet1.pdf

# Abstractions of a network connection



# Packet-oriented and stream-oriented connections

- UDP:  User Datagram Protocol
  - Unreliable, discrete packets of data
- TCP:  Transmission Control Protocol
  - Reliable data stream

# Internet addresses and sockets

- For IP version 4 (IPv4) host address is a 4-byte number
  - e.g. 127.0.0.1
  - Hostnames mapped to host IP addresses via DNS
  - ~4 billion distinct addresses
- Port is a 16-bit number (0-65535)
  - Assigned conventionally
    - e.g., port 80 is the standard port for web servers

# Networking in Java

- The java.net.InetAddress:

```
static InetAddress getByName(String host);
static InetAddress getByAddress(byte[] b);
static InetAddress getLocalHost();
```

- The java.net.Socket:

```
Socket(InetAddress addr, int port);
boolean      isConnected();
boolean      isClosed();
void         close();
InputStream  getInputStream();
OutputStream getOutputStream();
```

- The java.net.ServerSocket:

```
ServerSocket(int port);
Socket       accept();
void         close();
```

…

# Simple sockets demos

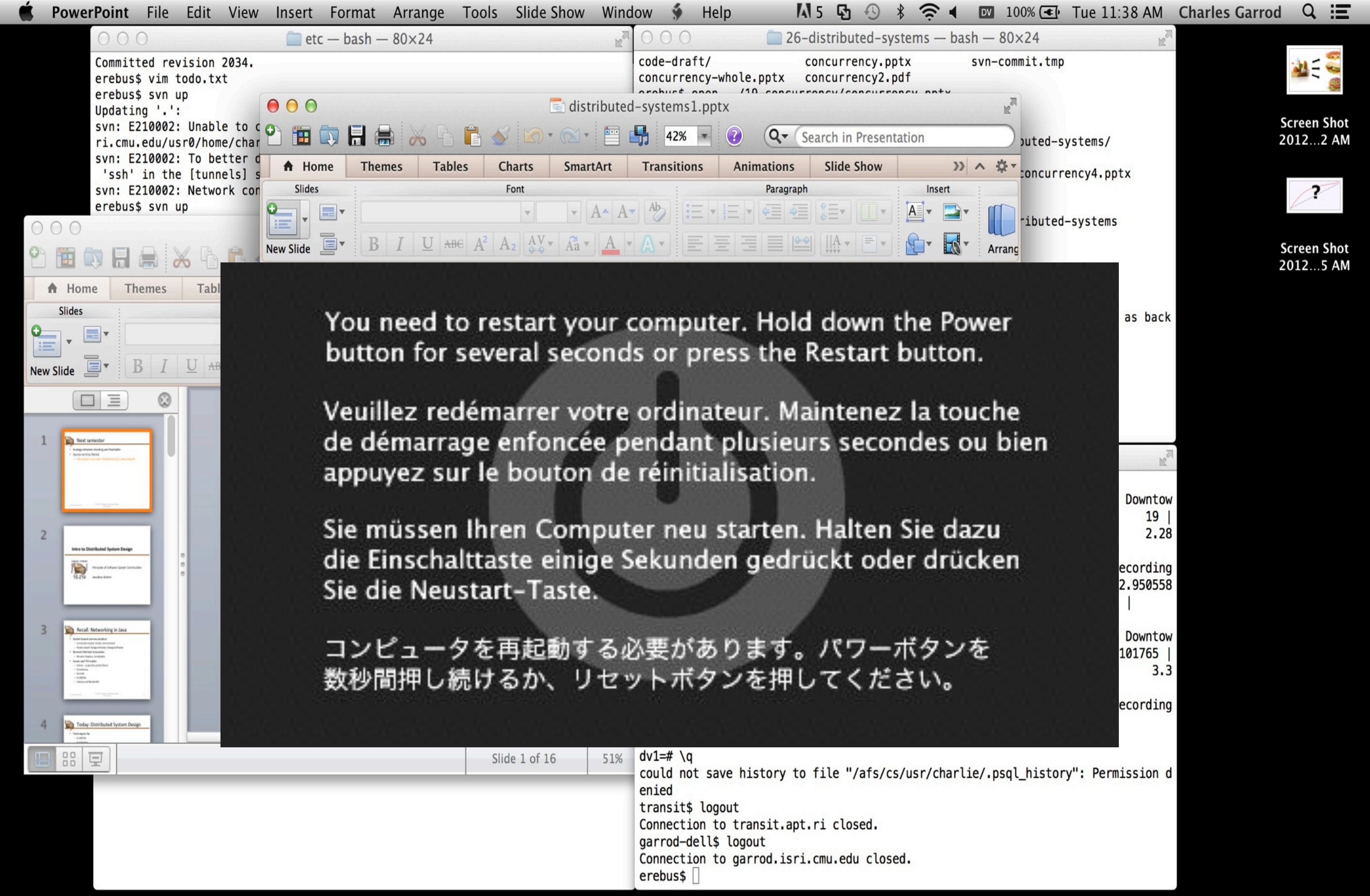
- NetworkServer.java
- A basic chat system:
  - TransferThread.java
  - TextSocketClient.java
  - TextSocketServer.java

# Higher levels of abstraction

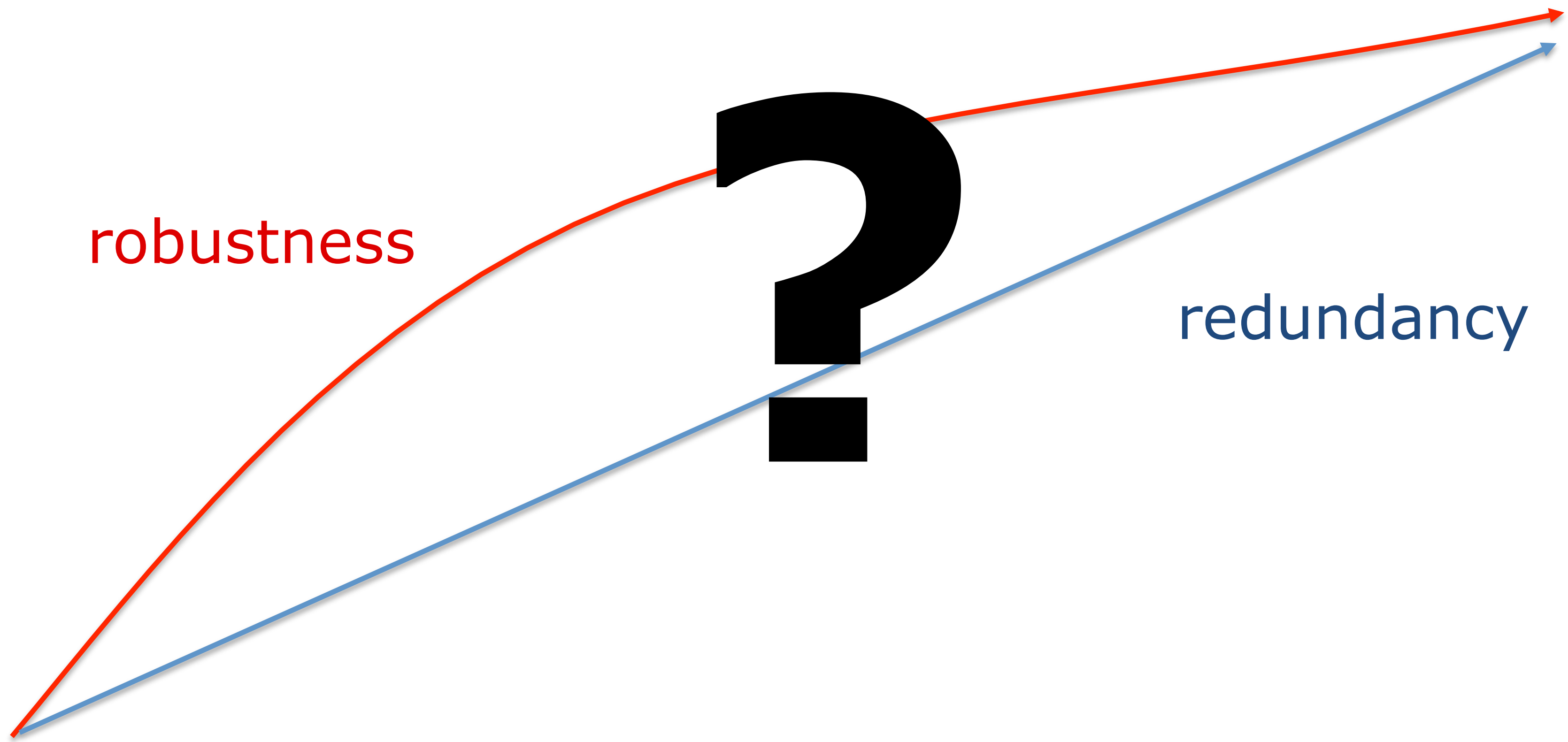
- Application-level communication protocols
- Frameworks for simple distributed computation
  - Remote Procedure Call (RPC)
  - Java Remote Method Invocation (RMI)
- Common patterns of distributed system design
- Complex computational frameworks
  - e.g., distributed map-reduce

# Today

- Java networking fundamentals
- Introduction to distributed systems
  - Motivation: reliability and scalability
  - Failure models
  - Techniques for:
    - Reliability (availability)
    - Scalability
    - Consistency


PowerPoint File Edit View Insert Format Arrange Tools Slide Show Window Help
Tue 11:38 AM Charles Garrod
etc — bash — 80×24
Committed revision 2034.
erebus$ vim todo.txt
erebus$ svn up
Updating '.':
erebus$ svn up
26-distributed-systems — bash — 80×24
code-draft/ concurrency.pptx svn-commit.tmp
concurrency-whole.pptx concurrency2.pdf
distributed-systems1.pptx
Home Themes Tables Charts SmartArt Transitions Animations Slide Show
Search in Presentation
Slides Font Paragraph Insert
New Slide
Screen Shot 2012...2 AM
Screen Shot 2012...5 AM
You need to restart your computer. Hold down the Power button for several seconds or press the Restart button.
Veuillez redémarrer votre ordinateur. Maintenez la touche de démarrage enfoncée pendant plusieurs secondes ou bien appuyez sur le bouton de réinitialisation.
Sie müssen Ihren Computer neu starten. Halten Sie dazu die Einschalttaste einige Sekunden gedrückt oder drücken Sie die Neustart-Taste.
コンピュータを再起動する必要があります。パワーボタンを数秒間押し続けるか、リセットボタンを押してください。
Slide 1 of 16
51%
dv1=# \q
could not save history to file "/afs/cs/usr/charlie/.psql_history": Permission denied
transit$ logout
Connection to transit.apt.ri closed.
garrod-dell$ logout
Connection to garrod.isri.cmu.edu closed.
erebus$

# Aside: The robustness vs. redundancy curve

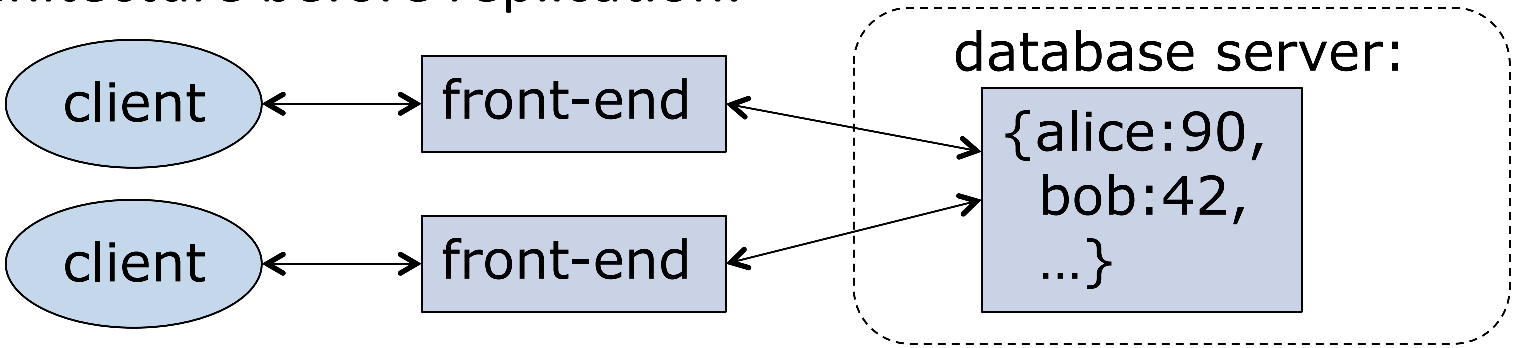


# Metrics of success

- Reliability
  - Often in terms of availability: fraction of time system is working
    - 99.999% available is "5 nines of availability"
- Scalability
  - Ability to handle workload growth

# A case study: Passive primary-backup replication

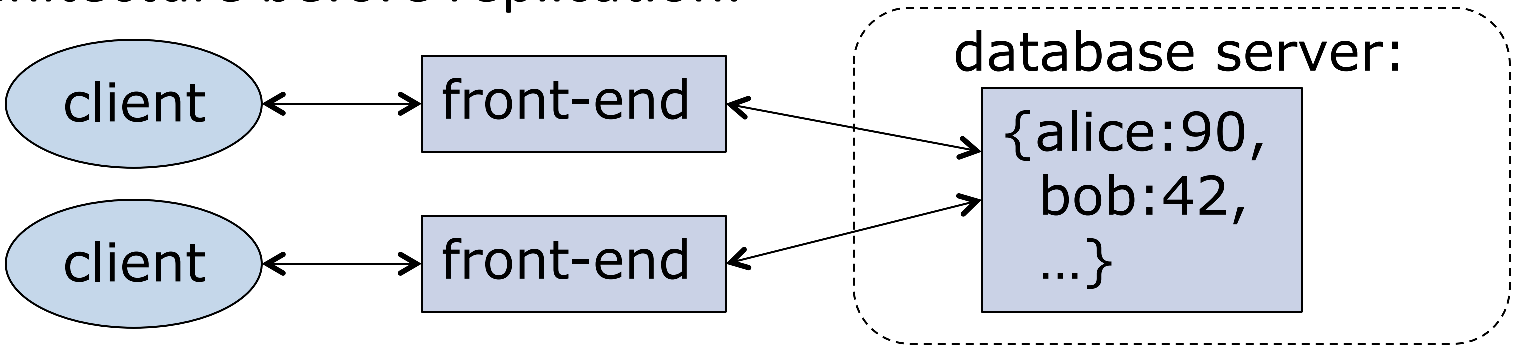
- Architecture before replication:




  – Problem: Database server might fail

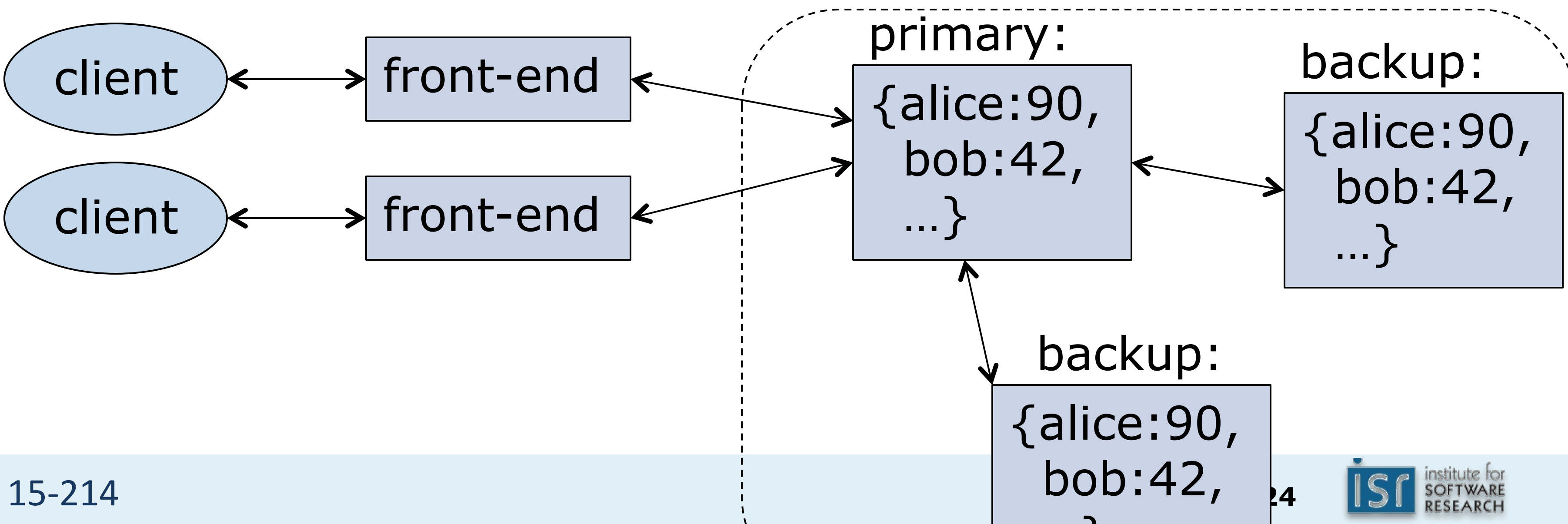
# A case study: Passive primary-backup replication

- Architecture before replication:




  – Problem: Database server might fail

- Solution: Replicate data onto multiple servers



# Passive primary-backup replication protocol

1. Front-end issues request with unique ID to primary DB
2. Primary checks request ID
   – If already executed request, re-send response and exit protocol
3. Primary executes request and stores response
4. If request is an update, primary DB sends updated state, ID, and response to all backups
   – Each backup sends an acknowledgement
5. After receiving all acknowledgements, primary DB sends response to front-end

# Issues with passive primary-backup replication

- If primary DB crashes, front-ends need to agree upon which unique backup is new primary DB
  - Primary failure vs. network failure?
- If backup DB becomes new primary, surviving replicas must agree on current DB state
- If backup DB crashes, primary must detect failure to remove the backup from the cluster
  - Backup failure vs. network failure?
- If replica fails* and recovers, it must detect that it previously failed
- Many subtle issues with partial failures
- …

# More issues…

- Concurrency problems?
  - Out of order message delivery?
    - Time…
- Performance problems?
  - 2n messages for n replicas
  - Failure of any replica can delay response
  - Routine network problems can delay response
- Scalability problems?
  - All replicas are written for each update
  - Primary DB responds to every request

# Types of failure behaviors

- Fail-stop
- Other halting failures
- Communication failures
  - Send/receive omissions
  - Network partitions
  - Message corruption
- Data corruption
- Performance failures
  - High packet loss rate
  - Low throughput
  - High latency
- Byzantine failures

# Common assumptions about failures

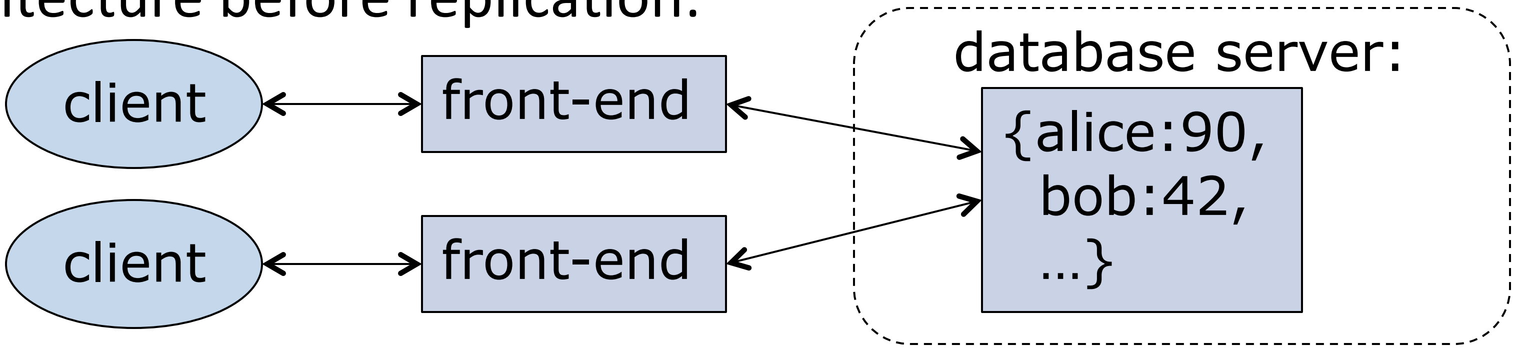
- Behavior of others is fail-stop (ugh)
- Network is reliable (ugh)
- Network is semi-reliable but asynchronous
- Network is lossy but messages are not corrupt
- Network failures are transitive
- Failures are independent
- Local data is not corrupt
- Failures are reliably detectable
- Failures are unreliably detectable

# Some distributed system design goals

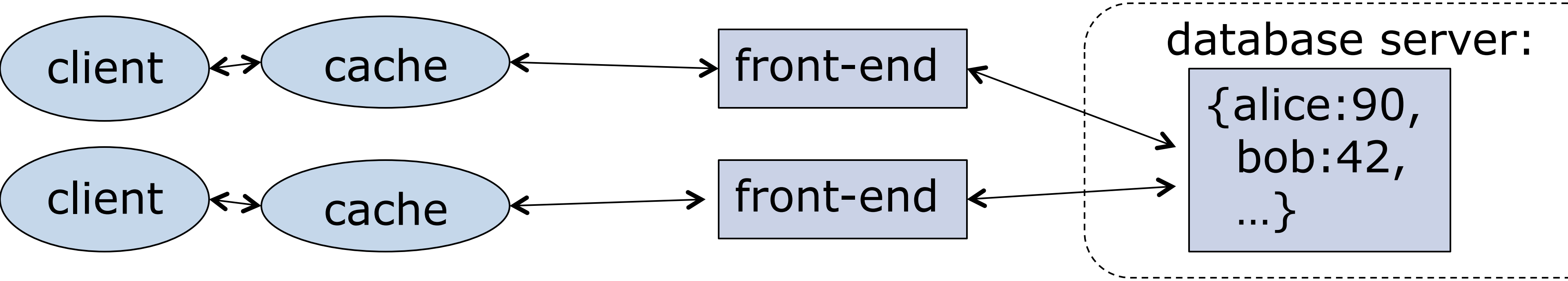
- The end-to-end principle
  - When possible, implement functionality at the end nodes (rather than the middle nodes) of a distributed system
- The robustness principle
  - Be strict in what you send, but be liberal in what you accept from others
    - Protocols
    - Failure behaviors
- Benefit from incremental changes
- Be redundant
  - Data replication
  - Checks for correctness

# Replication for scalability: Client-side caching

- Architecture before replication:

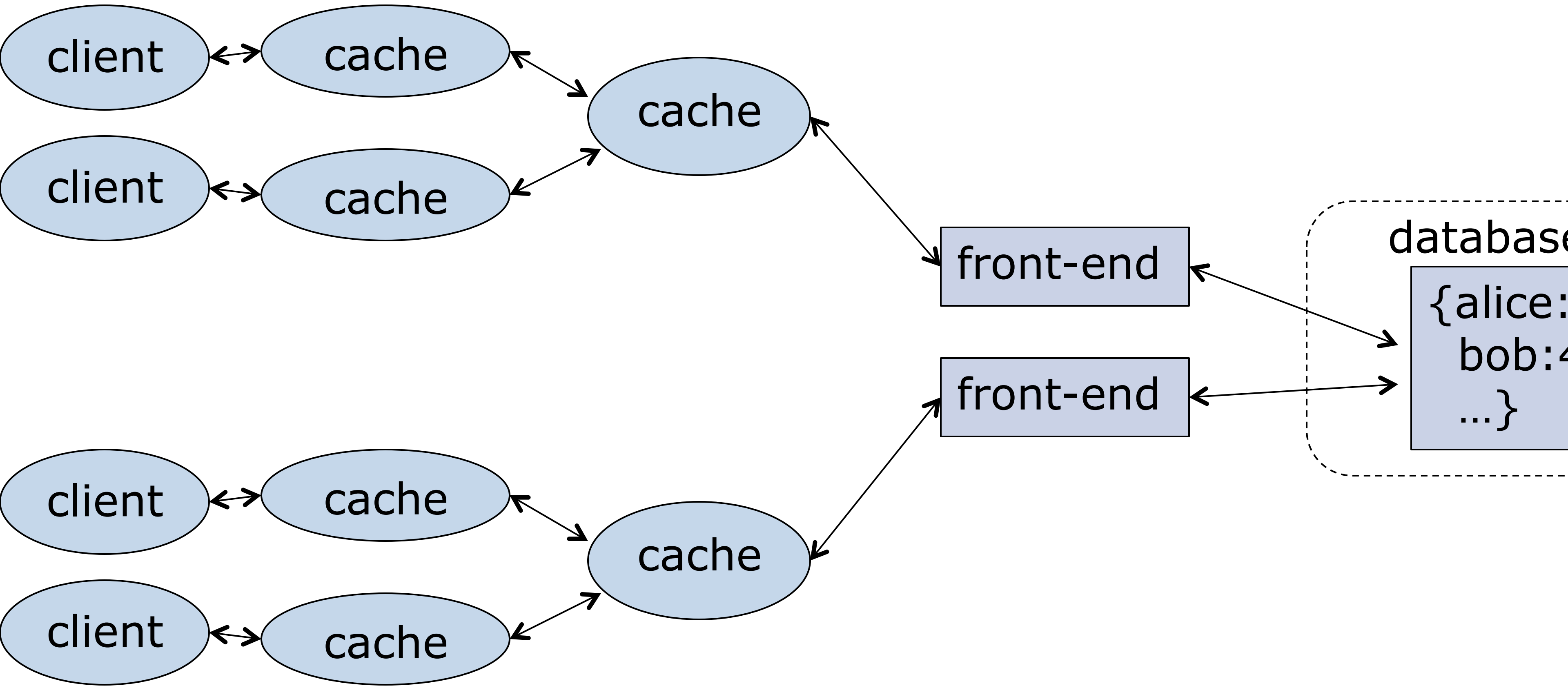



  - Problem: Server throughput is too low
- Solution: Cache responses at (or near) the client
  - Cache can respond to repeated read requests



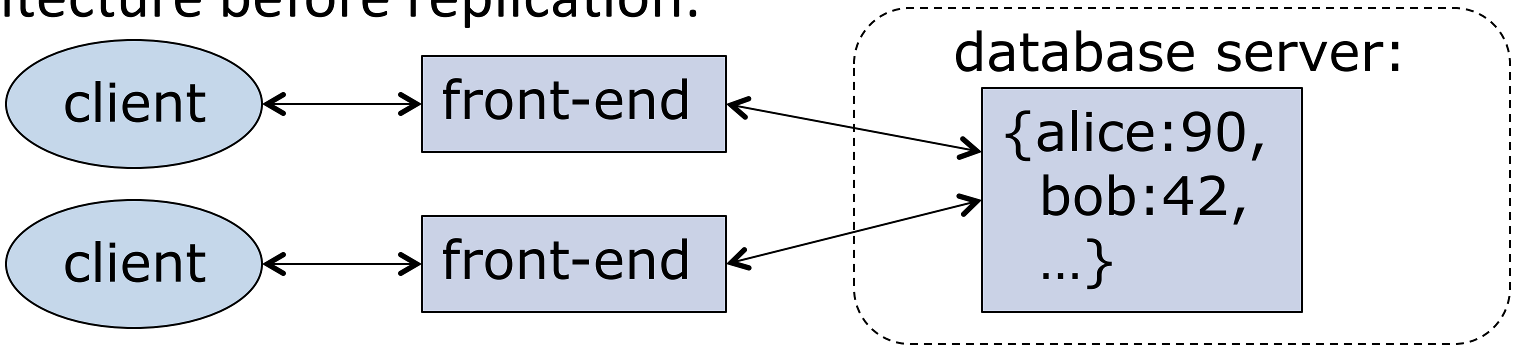
# Replication for scalability: Client-side caching

- Hierarchical client-side caches:



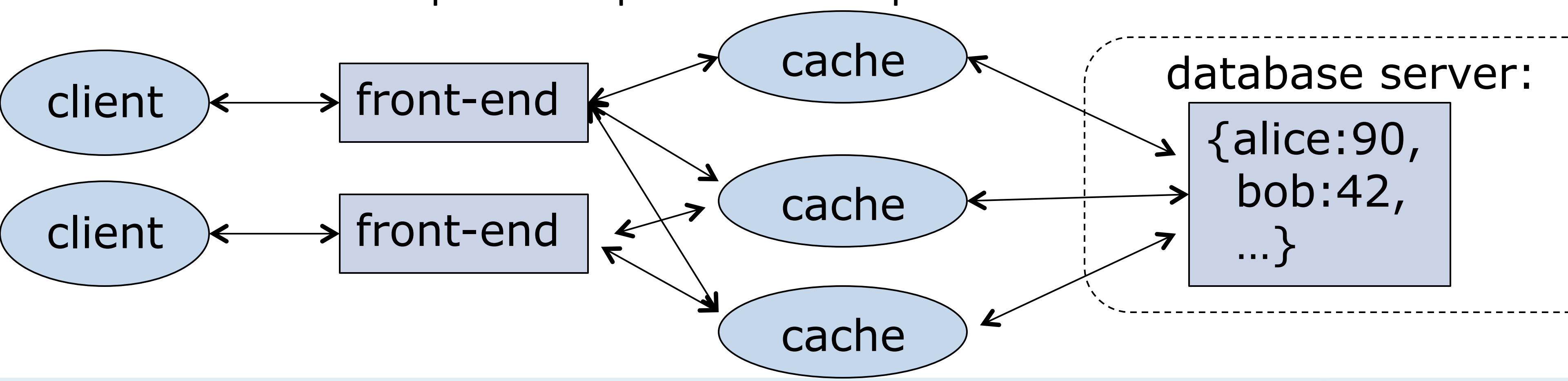
# Replication for scalability: Server-side caching

- Architecture before replication:




  – Problem: Database server throughput is too low

- Solution: Cache responses on multiple servers
  – Cache can respond to repeated read requests



# Cache invalidation

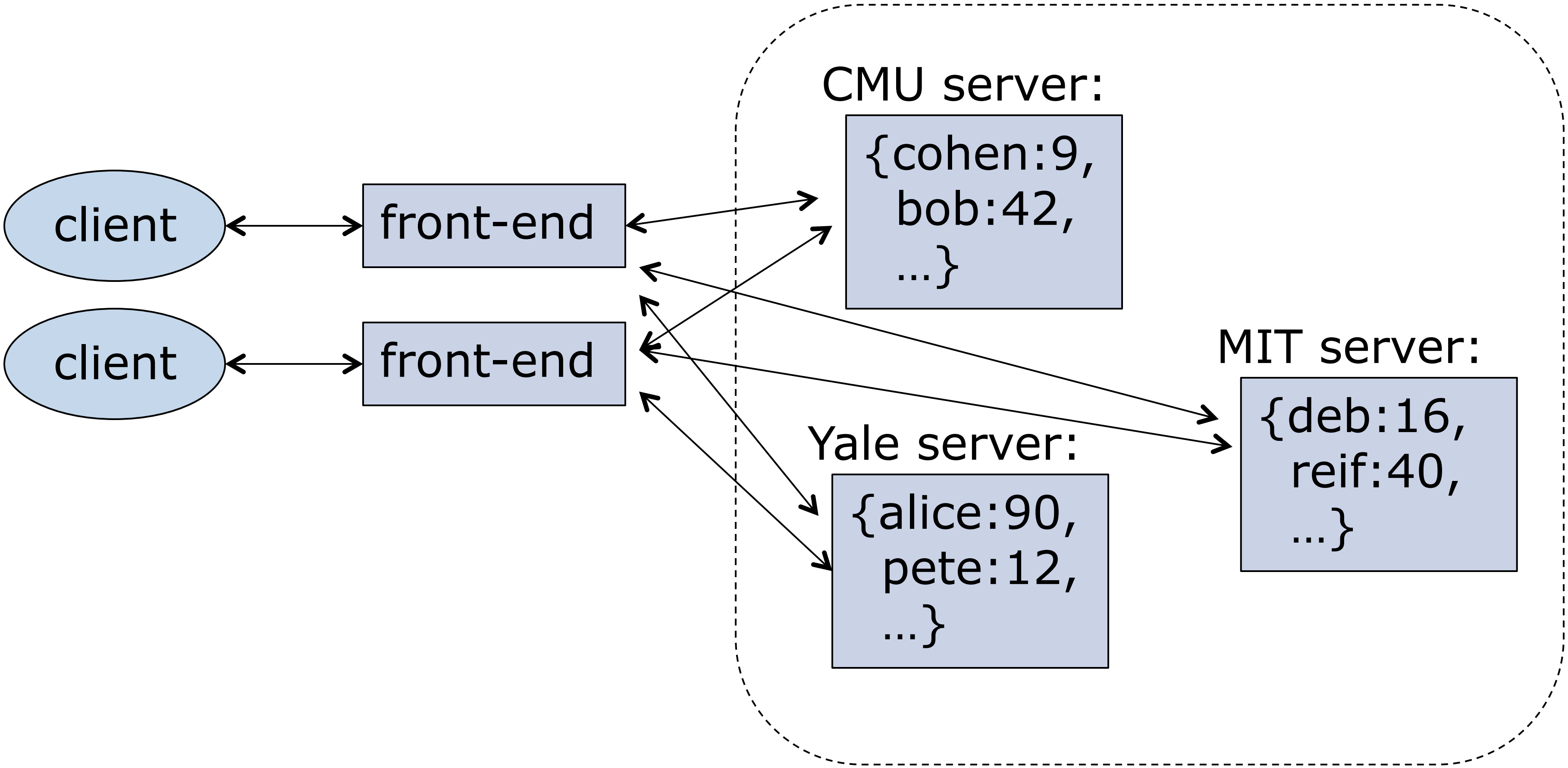
- Time-based invalidation  (a.k.a. expiration)
  - Read-any, write-one
  - Old cache entries automatically discarded
  - No expiration date needed for read-only data
- Update-based invalidation
  - Read-any, write-all
  - DB server broadcasts invalidation message to all caches when the DB is updated

# Cache replacement policies

- Problem:  caches have finite size
- Common* replacement policies
  - Optimal (Belady's) policy
    - Discard item not needed for longest time in future
  - Least Recently Used (LRU)
    - Track time of previous access, discard item accessed least recently
  - Least Frequently Used (LFU)
    - Count # times item is accessed, discard item accessed least frequently
  - Random
    - Discard a random item from the cache

# Partitioning for scalability

- Partition data based on some property, put each partition on a different server



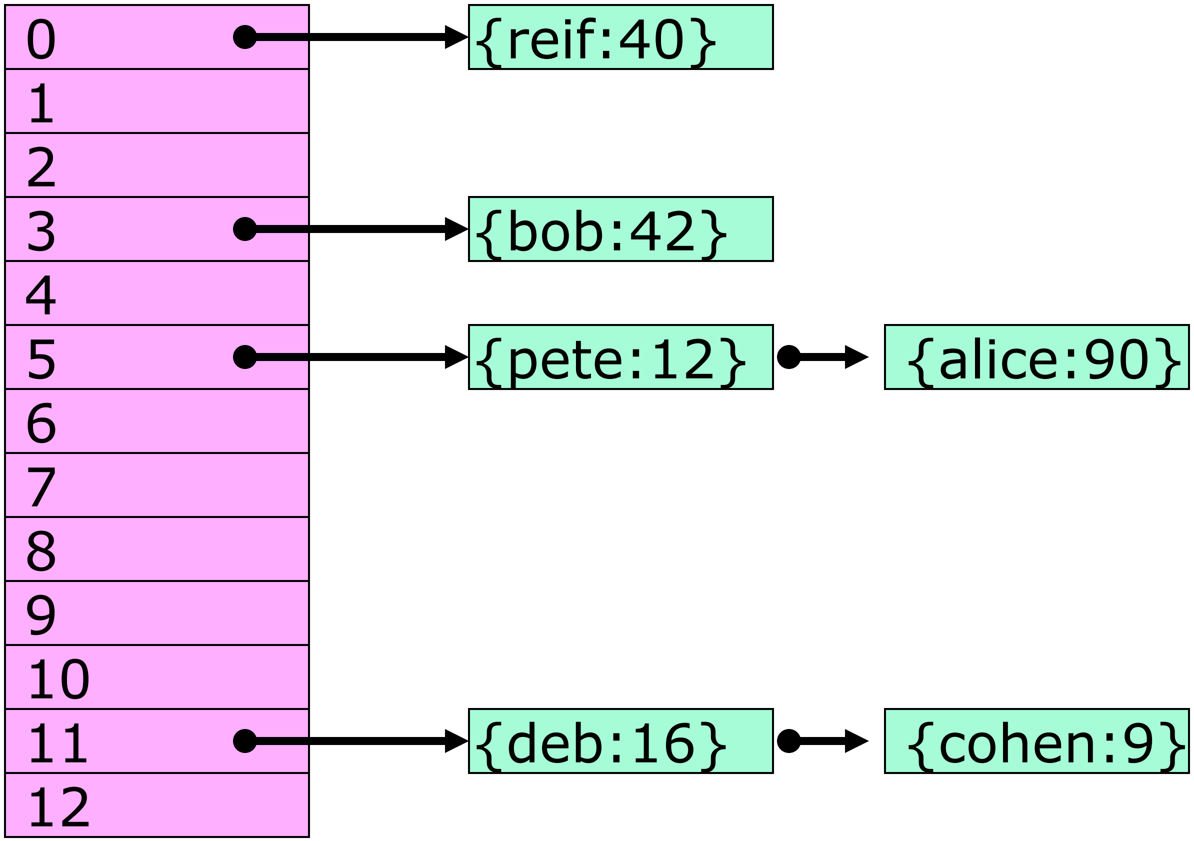
# Horizontal partitioning

- a.k.a. "sharding"
- A table of data:

| username | school | value |
| --- | --- | --- |
| cohen | CMU | 9 |
| bob | CMU | 42 |
| alice | Yale | 90 |
| pete | Yale | 12 |
| deb | MIT | 16 |
| reif | MIT | 40 |

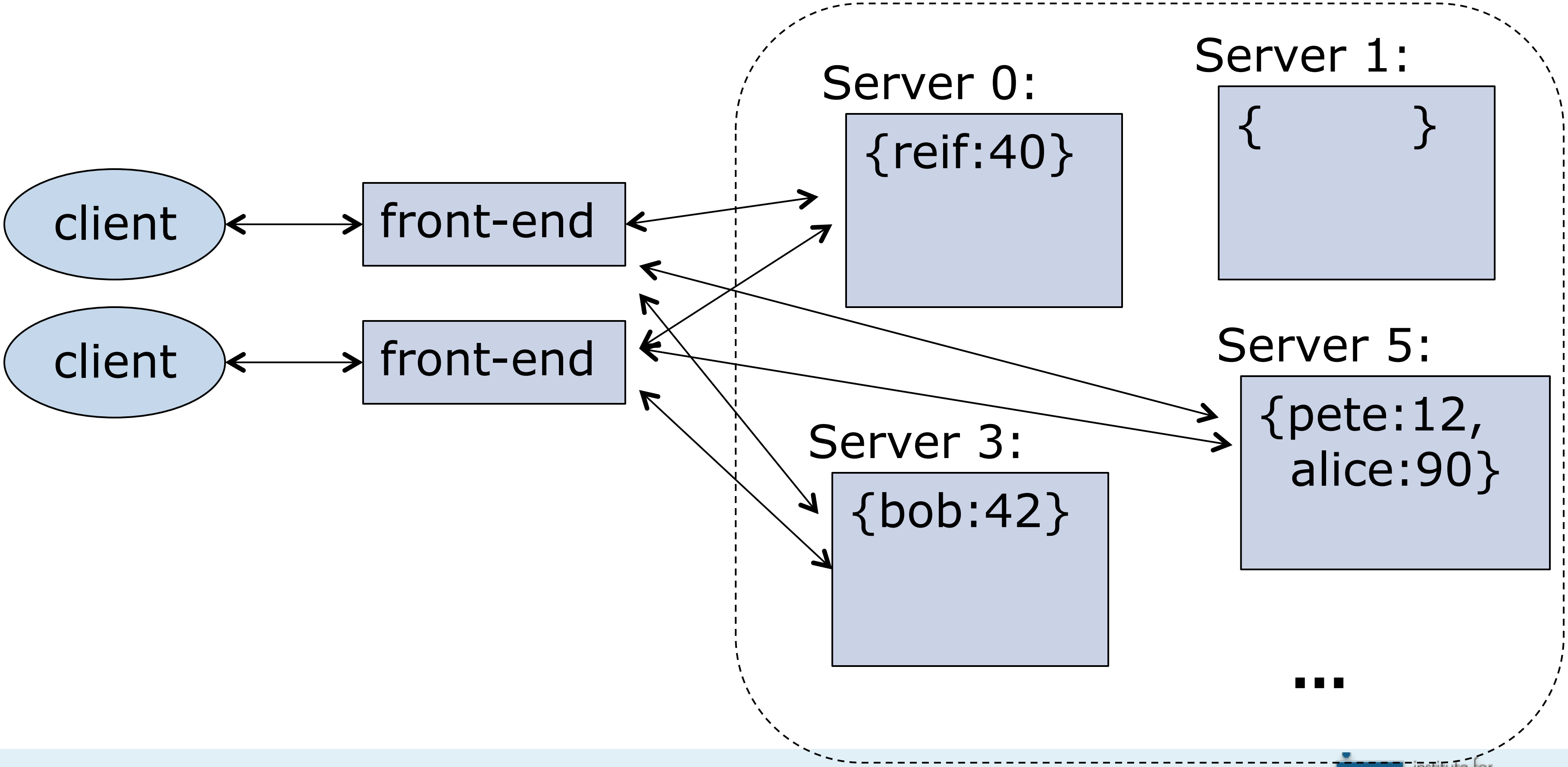
# Recall: Basic hash tables

- For `n`-size hash table, put each item `X` in the bucket: `X.hashCode() % n`



# Partitioning with a distributed hash table

- Each server stores data for one bucket
- To store or retrieve an item, front-end server hashes the key, contacts the server storing that bucket



# Consistent hashing

- Goal: Benefit from incremental changes
  - Resizing the hash table (i.e., adding or removing a server) should not require moving many objects
- E.g., Interpret the range of hash codes as a ring
  - Each bucket stores data for a range of the ring
    - Assign each bucket an ID in the range of hash codes
    - To store item `X` don't compute `X.hashCode() % n`. Instead, place `X` in bucket with the same ID as or next higher ID than `X.hashCode()`

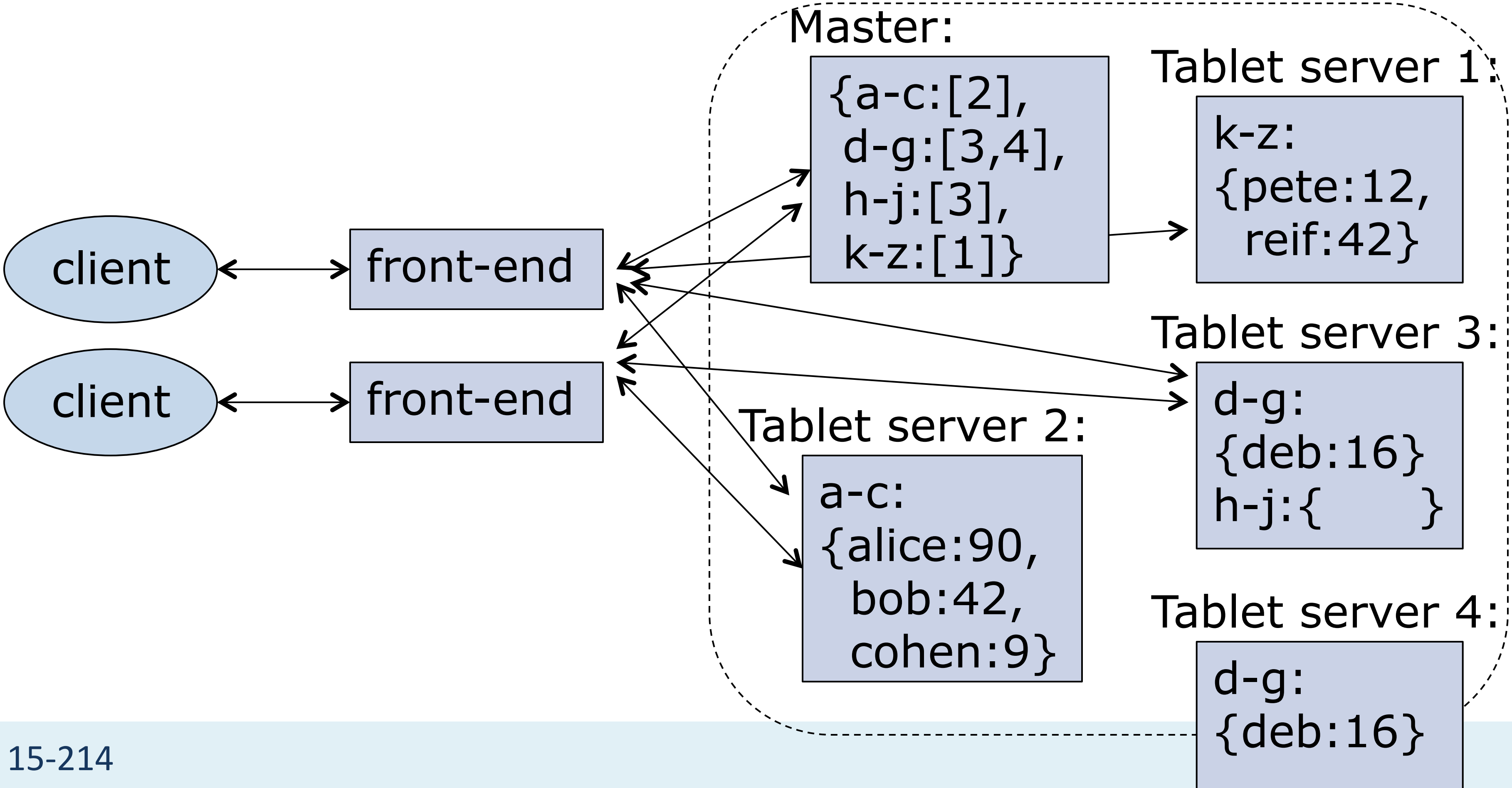
# Problems with hash-based partitioning

- Front-ends need to determine server for each bucket
  - Each front-end stores look-up table?
  - Master server storing look-up table?
  - Routing-based approaches?
- Places related content on different servers
  - Consider *range* queries:

```
SELECT * FROM users WHERE lastname STARTSWITH 'G'
```

# Master/tablet-based systems

- Dynamically allocate range-based partitions
  - Master server maintains tablet-to-server assignments
  - Tablet servers store actual data
  - Front-ends cache tablet-to-server assignments



# Coming next…

- More distributed systems
  - MapReduce