Typical Cloud Computing Services

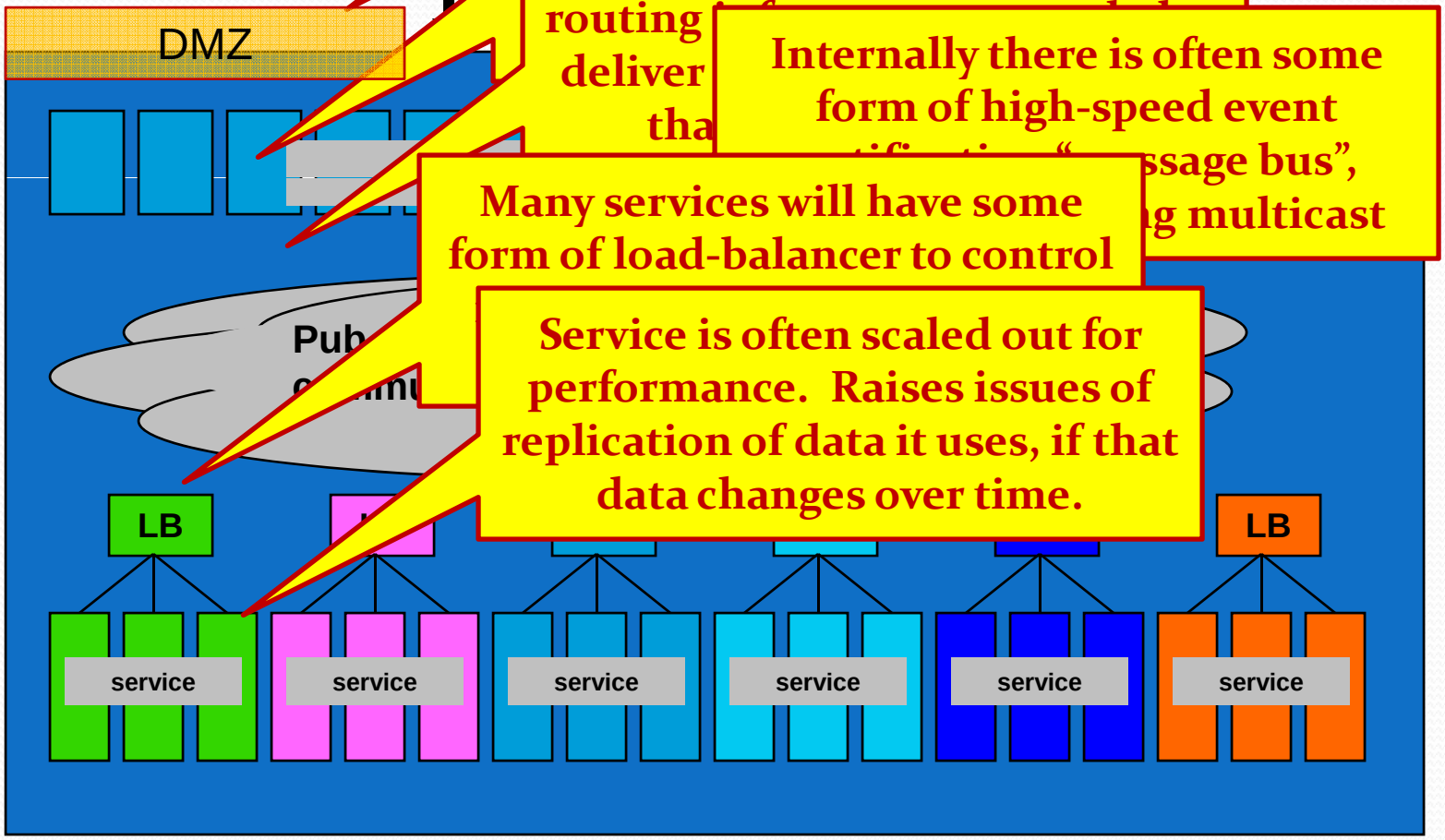
Ken Birman

Cornell University. CS5410 Fall 2008.

Last time: standards…

- We looked mostly at big architectural standards
- But there are also standard ways to build cloud infrastructure support.
- Today: review many of the things one normally finds in a cloud computing setting, discuss what role each plays
  - Our goal *is not* to talk about best implementations yet
  - We'll do that later
  - Rather, focus on structure and roles and functionality

A glimpse in



If needed, machines in the "DMZ" (demilitarized zone) can

Either a server that builds web

Internal naming convention and routing deliver tha

Internally there is often some form of high-speed event ssage bus", g multicast

Many services will have some form of load-balancer to control

Service is often scaled out for performance. Raises issues of replication of data it uses, if that data changes over time.

More components

- Data center has a physical structure (racks of machines) and a logical structure (the one we just saw)
  - Something must map logical roles to physical machines
  - Must launch the applications needed on them
  - And then monitor them and relaunch if crashes ensue
  - Poses optimization challenges
- We probably have multiple data centers
  - Must control the external DNS, tell it how to route
  - Answer could differ for different clients

More components

- Our data center has a security infrastructure involving keys, certificates storing them, permissions
- Something may need to decide not just where to put services, but also which ones need to be up, and how replicated they should be
- Since server locations can vary and server group members change, we need to track this information and use it to adapt routing decisions
- The server instances need a way to be given parameters and environment data

More components

- Many kinds of events may need to be replicated
  - Parameter or configuration changes that force services to adapt themselves
  - Updates to the data used by the little service groups (which may not be so small…)
  - Major system-wide events, like “we’re being attacked!” or “Scotty, take us to Warp four”
- Leads to what are called event notification infrastructures, also called publish-subscribe systems or event queuing middleware systems

More components

- Status monitoring components
  - To detect failures and other big events
  - To help with performance tuning and adaptation
  - To assist in debugging
  - Even for routine load-balancing
- Load balancers (now that we're on that topic…)
  - Which need to know about loads and membership
  - But also may need to do deep packet inspection to look for things like session id's

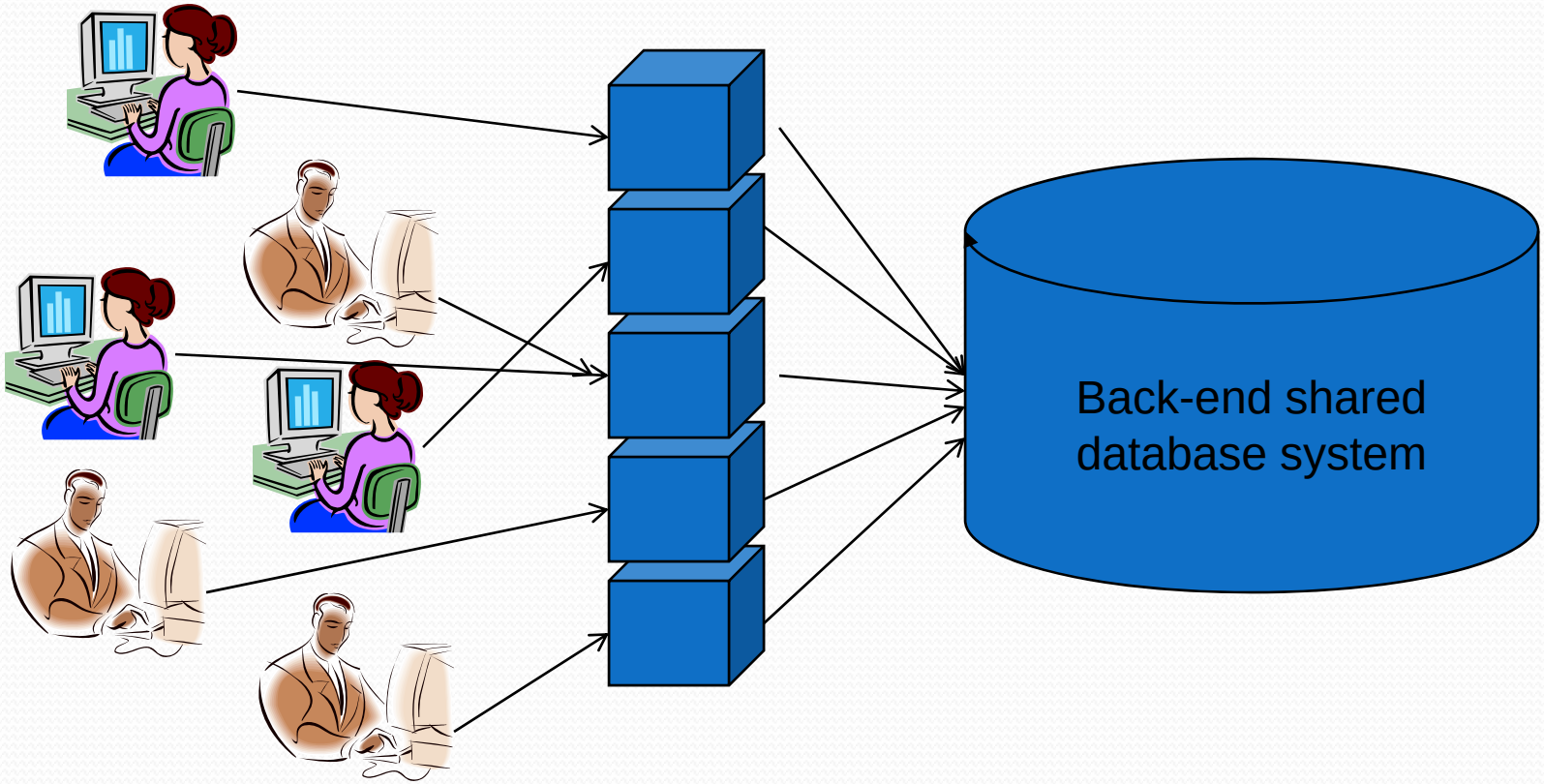
More, and more, and more…

- Locking service
  - Helps prevent concurrency conflicts, such as two services trying to create the identical file
- Global file system
  - Could be as simple as a normal networked file system, or as fancy as Google's GFS
- Databases
  - Often, these run on clusters with their own scaling solutions…

Let's drill down…

- Suppose one wanted to build an application that
  - Has some sort of "dynamic" state (receives updates)
  - Load-balances queries
  - Is fault-tolerant
- How would we do this?

Today's prevailing solution



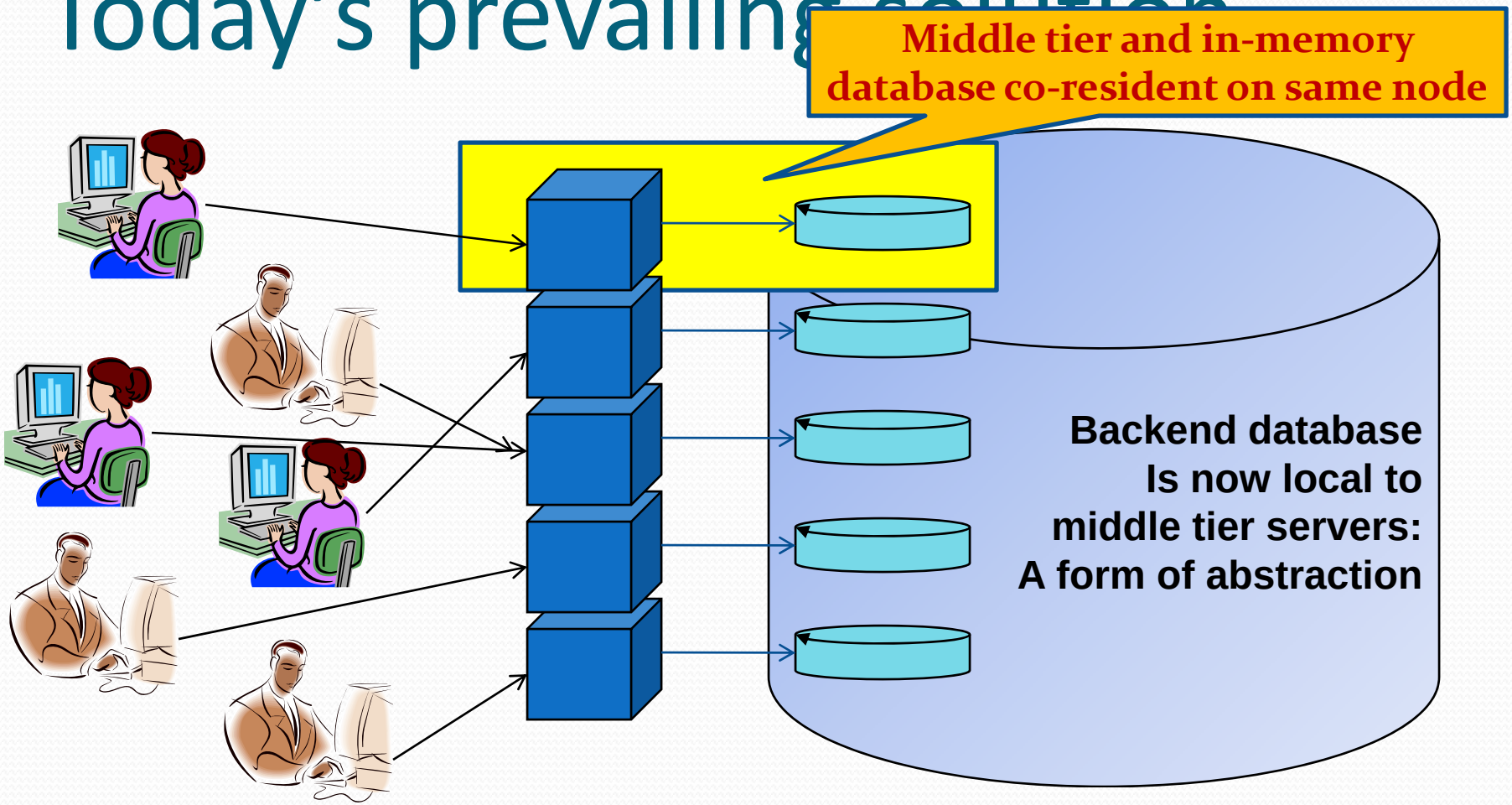
Concerns?

- Potentially slow (especially during failures)
- Doesn't work well for applications that don't split cleanly between "persistent" state (that can be stored in the database) and "business logic" (which has no persistent state)

Can we do better?

- What about some form of in-memory database
  - Could be a true database
  - Or it could be any other form of storage "local" to the business logic tier
- This eliminates the back-end database
  - More accurately, it replaces the single back-end with a set of local services, one per middle-tier node
  - This is a side-effect of the way that web services are defined: the middle-tier *must be stateless*
- But how can we build such a thing?

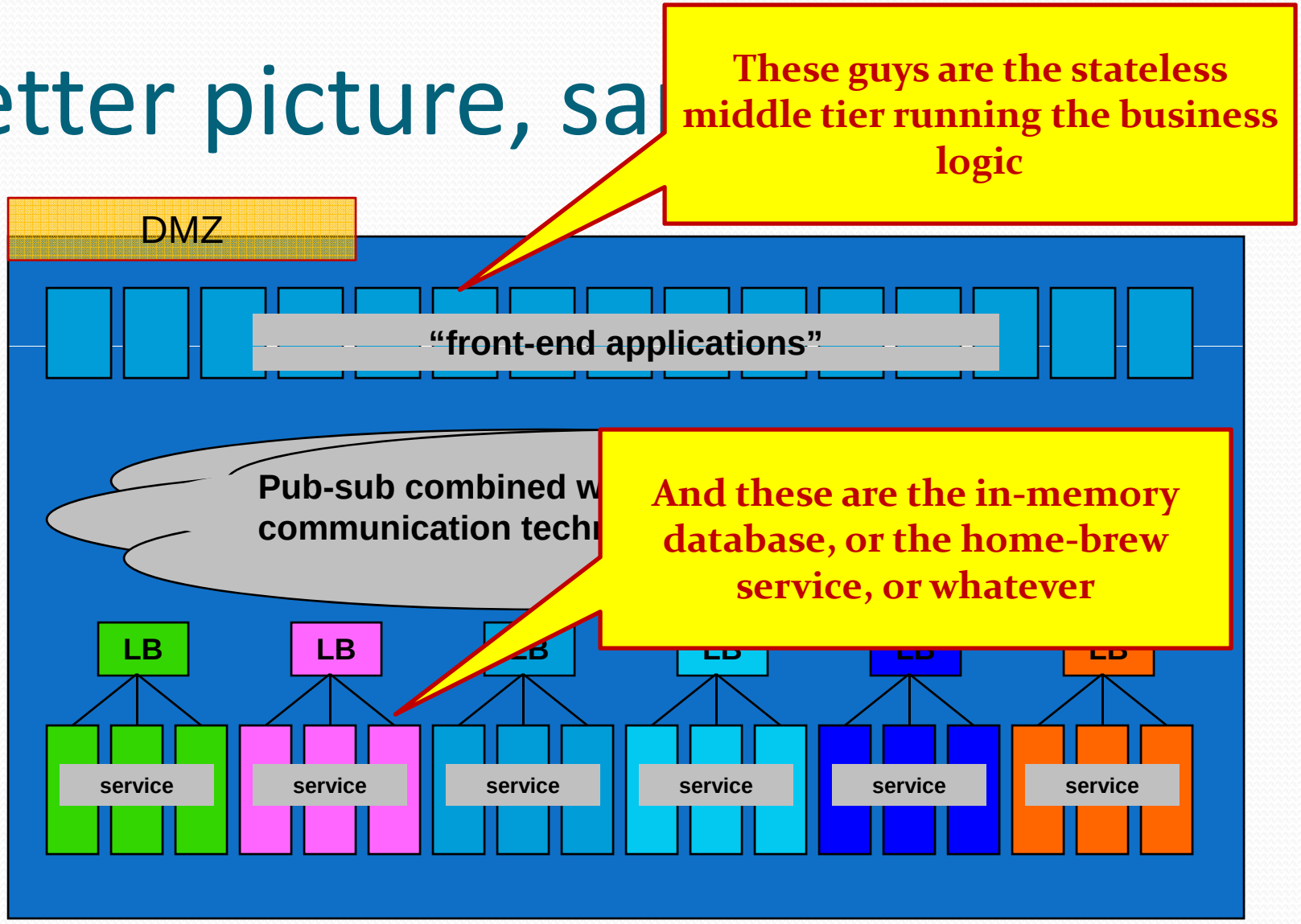
Today's prevailing solution



Services with in-memory state

- Really, several cases
  - We showed a stateless middle tier running business logic and talking to an in-memory database
  - But in our datacenter architecture, the stateless tier was "on top" and we might need to implement replicated services of our very own, only some of which are databases or use them
  - So we should perhaps decouple the middle tier and not assume that every server instance has its very own middle tier partner….

Better picture, sa

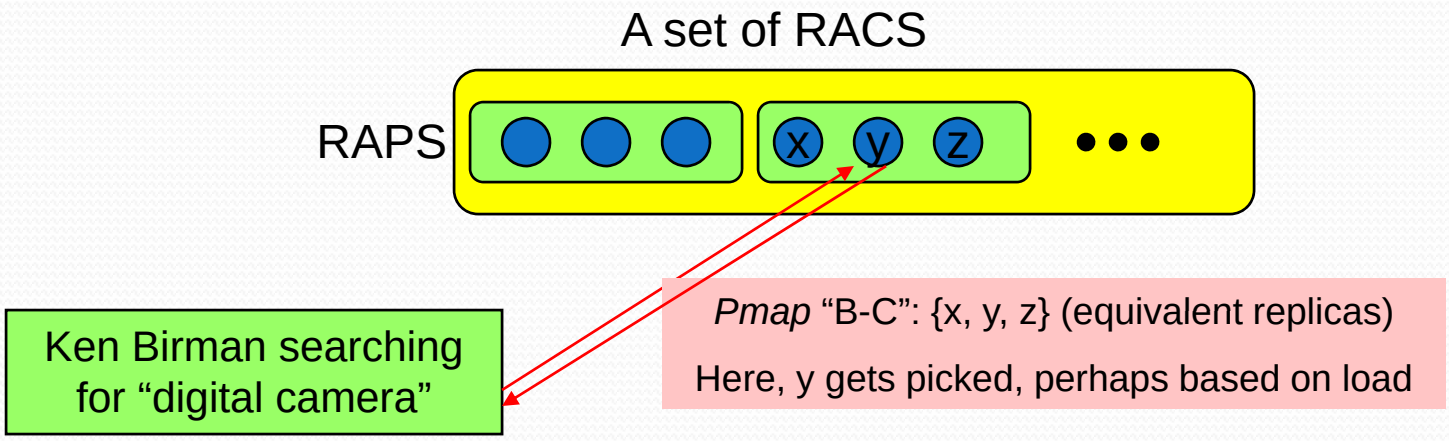


More load-spreading steps

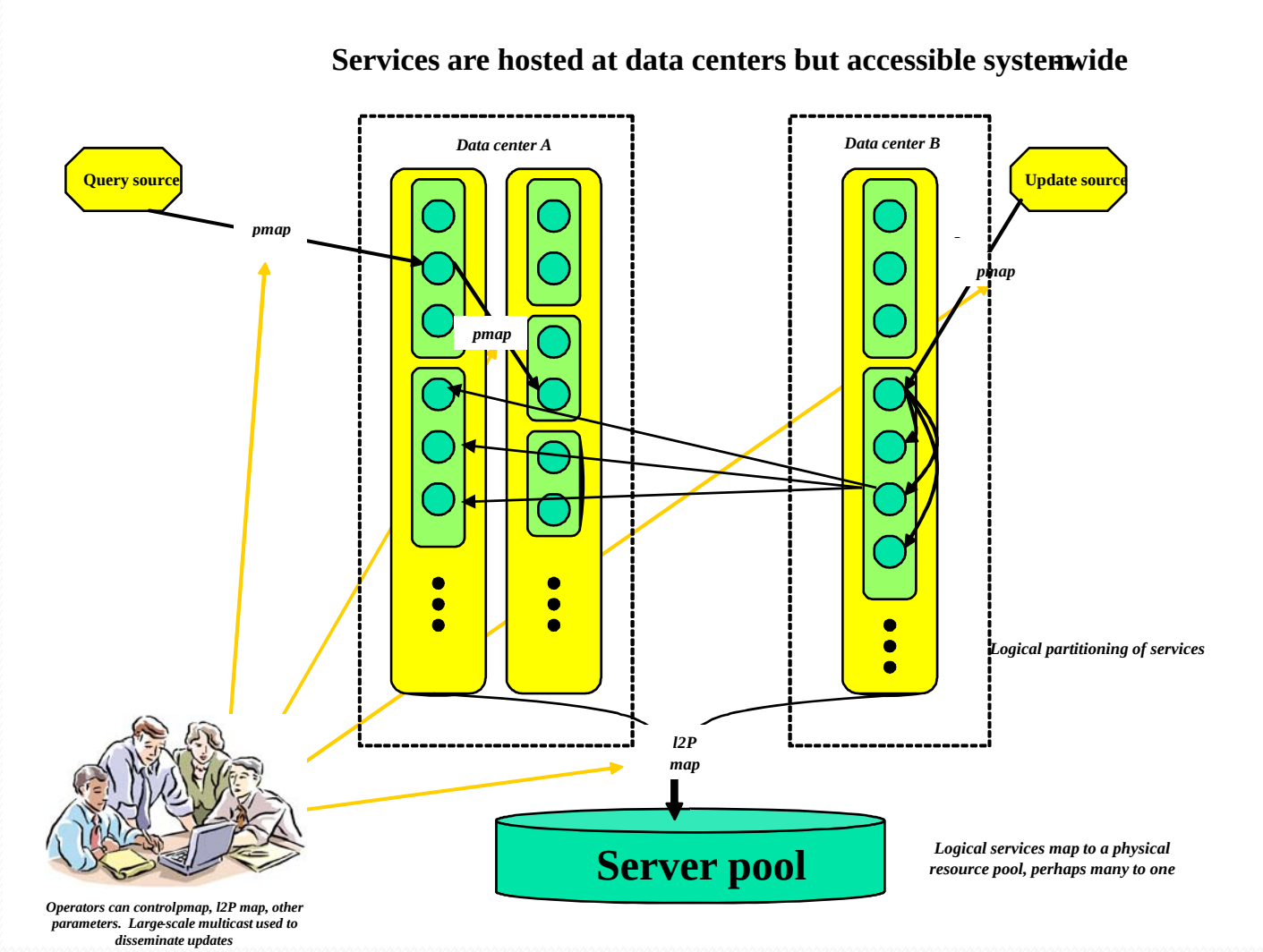
- If every server handles all the associated data…
  - Then if the underlying data changes, every server needs to see every update
  - For example, in an inventory service, the data would be the inventory for a given kind of thing, like a book.
  - Updates would occur when the book is sold or restocked
- Obvious idea: partition the database so that groups of servers handle just a part of the inventory (or whatever)
  - Router needs to be able to extract keys from request: another need for “deep packet inspection” in routers

A RAPS of RACS (Jim Gray)

- RAPS: A reliable array of partitioned subservices
- RACS: A reliable array of cloned server processes



RAPS of RACS in Data Centers

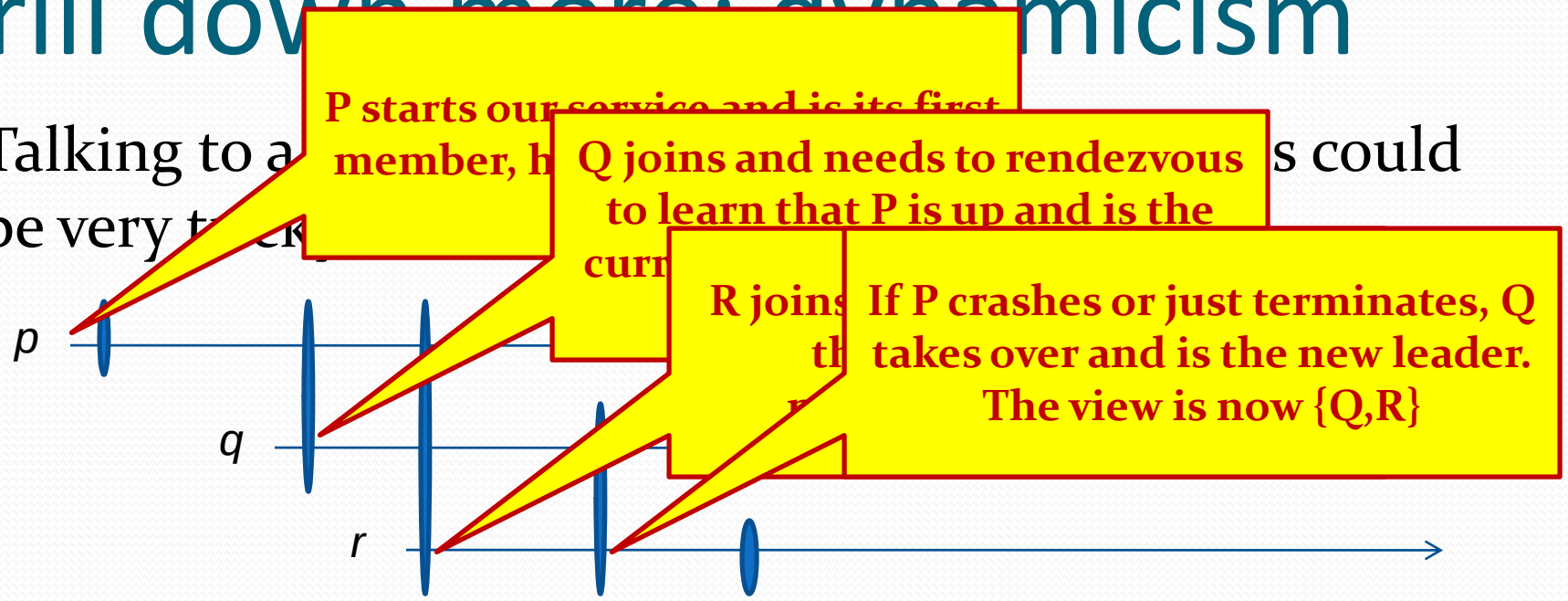


Partitioning increases challenge

- Previously, routing to a server was just a question of finding some representative of the server
  - A kind of “anycast”
- But now, in a service-specific way, need to
  - Extract the partitioning key (different services will have different notions of what this means!)
  - Figure out who currently handles that key
  - Send it to the right server instance (RAPS)
  - Do so in a way that works even if the RAPS membership is changing when we do it!

Drill dow[illegible]micism

- Talking to a[illegible]s could be very t[illegible]



- The client system will probably get "old" mapping data
- Hence may try and talk to p when the service is being represented by q, or r…

Causes of dynamicism ("churn")?

- Changing load patterns
- Failures
- Routine system maintenance, like disk upgrades or even swapping one cluster out and another one in
- At Google, Amazon this is a *continuous process!*
  - In the OSDI paper on Map Reduce, authors comment that during one experiment that involved 2000 nodes, sets of 80 kept dropping out.
  - Google had their machines in racks of 20, 4 per power unit, so this makes perfect sense: power upgrades…

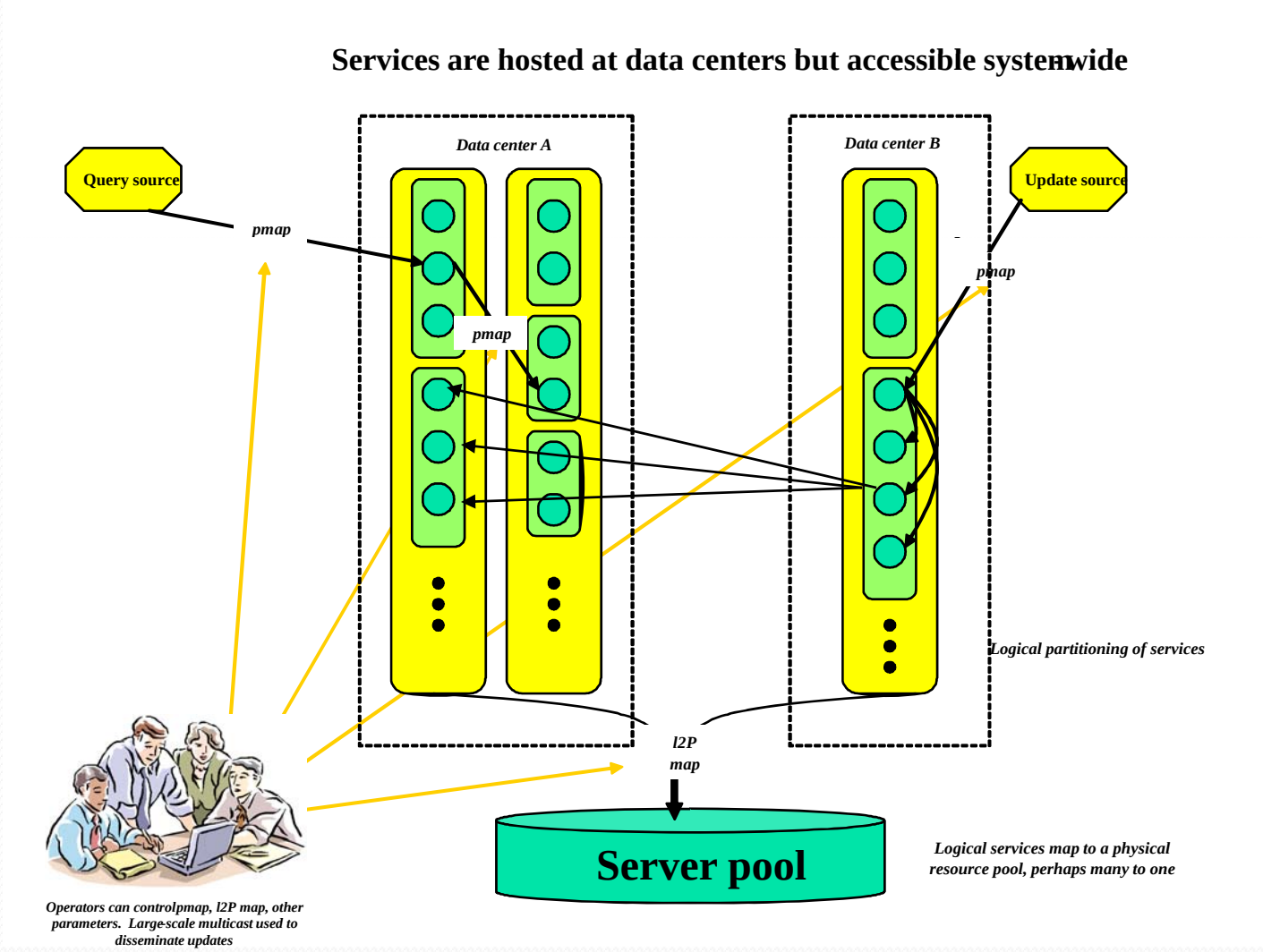
Causes of dynamicism

- IBM team that built DCS describes a "whiteboard" application used internal to their system
  - Information used by the system, updated by the system
  - Organized as shared pages, like Wiki pages, but updated under application control
- They observed
  - Tremendous variance in the sets of applications monitoring each page (each topic, if you wish)
  - High update rates
  - **Tens of thousands of membership events per second!**

Causes of dynamicism

- One version of the Amazon.com architecture used publish-subscribe products for all interactions between front-end and back-end servers
- They created pub-sub topics very casually
  - In fact, each client "session" had its own pub-sub topic
  - And each request created a unique reply "topic"
- Goal was to make it easy to monitor/debug by listening in… but effect was to create huge rate of membership changes in routing infrastructure
- **Again, tens of thousands per second!**

Revisit our RAPS of RACS… but now think of the sets as changing constantly



Implications of dynamics?

- How can we conceal this turbulence so that clients of our system won't experience disruption?
  - We'll look closely at this topic soon, but not right away
  - Requires several lectures on the topic of "dynamic group membership"
- How do implement things like routing
  - At a minimum, need to use our event notification infrastructure to tell everyone who might need to know
- Poses a theoretical question too
  - When can a highly dynamic system mimic a "static" one?

Recall our original goal…

- We're seeing that "membership tracking" in our data center is more of a problem that it originally seemed
  - We've posed a kind of theory question (can we mimic a static system
  - But introduced huge sources of membership dynamics
  - Not to mention failures, load changes that induce reconfiguration to handle new request patterns
- Plus, beyond tracking changes, need ways to program the internal routing infrastructure so that requests will reach the right nodes

One sample challenge problem

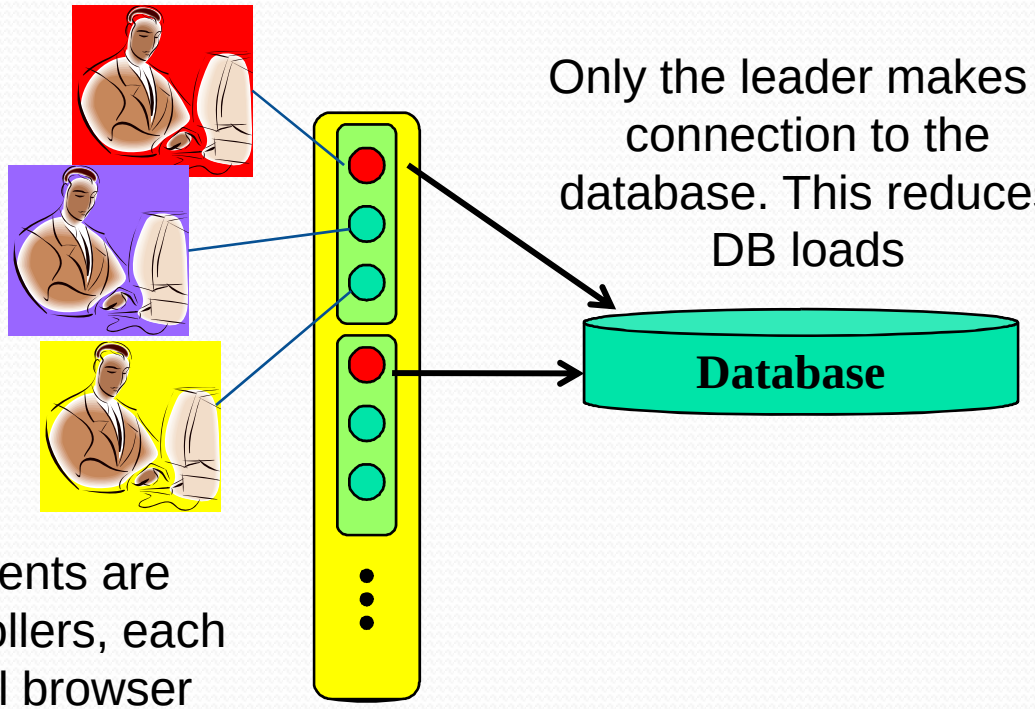
- Are these questions hard to solve? Let's tackle one
- Consider a service (a single RACS if you wish)
  - Might have no members (not running)
  - One member (just launched…)
  - Many members (steady state…)
  - … and changes may happen rapidly
- And let's assign a special role to one member
  - Call it the leader

Who needs leaders?

- One real example: In French ATC data center, each ATC sector is managed by a small group of controllers
  - The "group" (RACS) has one agent on each controller workstation, tracking actions by that person
  - They back one-another up, but normally have distinct roles. One guys directs the planes, one plans routes, etc
- There is a shared back-end database, and it can't handle huge numbers of connections
- So we have the leader connect to the database on behalf of the whole group

Leader connected to a database

-



Only the leader makes a connection to the database. This reduces DB loads

Data center clients are the ATC controllers, each using a special browser

Here's our RAPS of RAC but each RACS has a leader now (red node)

Other leader “roles”

- Leader might be in charge of updates to the group (for example, if the database reports a change). A leader might also monitor a sensor, or camera, or video feed and relay the data
- Leader can hold a “lock” of some sort, or perhaps only hold it initially (it would pass it to someone who makes a request, etc)
- Generalization of a leader is an agreed *ranking* of group members, very useful when subdividing tasks to perform them in a parallel manner

Challenges

- How to launch such a service?
  - Your application starts up… and should either become the leader if none is running, or join in if the service is up (and keep in mind: service may be "going down" right at the same time!)
  - How to rendezvous with it?
    - Could use UDP broadcasts ("Is anyone there?")
    - Or perhaps exploit the DNS? Register service name much like a virtual computer name – "inventory.pac-nw.amazon.com"
    - Could use a web service in the same role
    - Could ask a human to tell you (seems like a bad idea…)

Challenges

- Suppose p is the current leader and you are next in line
  - How did you know that you're next in line? ("ranking")
  - How to monitor p?
  - If p crashes, how to take over in an official way that won't cause confusion (no link to database… or two links…)
  - If p was only temporarily down, how will you deal with this?
  - What would you do if p and q start concurrently?
  - What if p is up, and q and r start concurrently?
  - What about failures *during the protocol?*

Homework 1

- To get your hands dirty, we want you to use Visual Studio to implement a (mostly) UDP-based solution to this problem, then evaluate it and hand in your code
- You'll do this working individually
- Evaluation will focus on scalability and performance
  - How long does it take to join the service, or to take over as a new leader if the old one unexpectedly crashes?
  - How does this scale as a function of the number of application groups on each machine *(if too hard can skip)*
  - Why is your solution correct?

Back to data center services

- We can see that the membership service within a data center is very complex and somewhat spread out
  - In effect, part of the communication infrastructure
  - Issues range from tracking changing membership and detecting failures to making sure that the routing system, load balancers, and clients know who to talk to
  - And now we're seeing that membership can have "semantics" such as rankings or leader roles
- This leads us towards concept of execution models for dynamic distributed systems

Organizing our technologies

- It makes sense to think in terms of layers:
  - Lowest layer has core Internet mechanisms, like DNS
    - We can control DNS mappings, but it isn't totally trivial…
  - Next layer has core services
    - Such as membership tracking, help launching services, replication tools, event notification, packet routing, load balancing, etc
  - Next layer has higher-level services that use the core
    - Network file system, Map/Reduce, overlay network for stream media delivery, distributed hash tables….
  - Applications reside "on top"

On Thursday?

- We'll peek inside of Map Reduce to see what it offers
  - An example of a powerful user-oriented tool
  - Map Reduce hides most of the complexities from clients, for a particular class of data center computing problems
  - It was built using infrastructure services of the kind we're discussing…
- To prepare for class, please read the Map Reduce paper
  - Short version from CACM (7 pages) or long version from OSDI (14 pages)
  - Links available on our course web page – click to the slides page and look at Thursday entry…