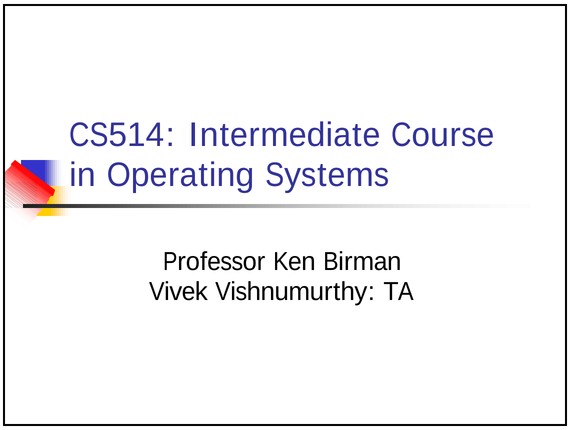
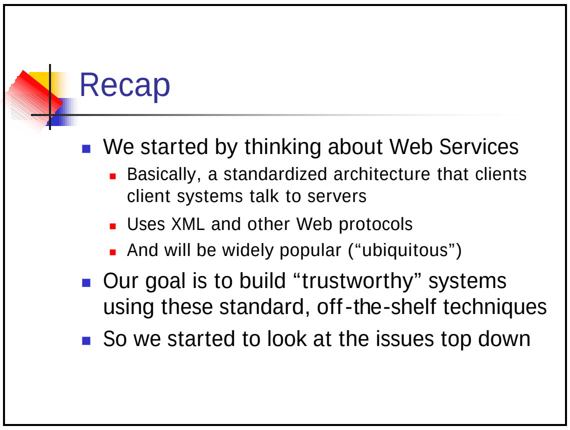


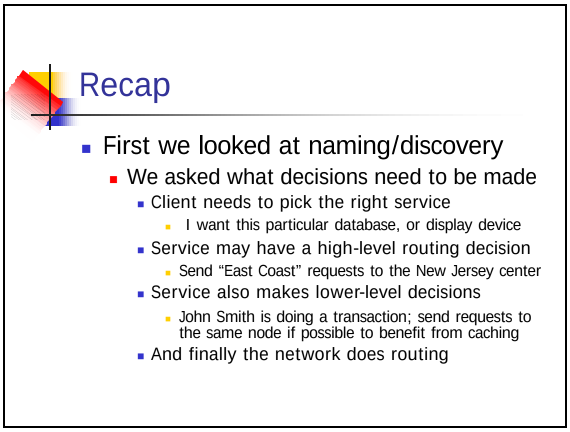
CS514: Intermediate Course in Operating Systems

Professor Ken Birman
Vivek Vishnumurthy: TA



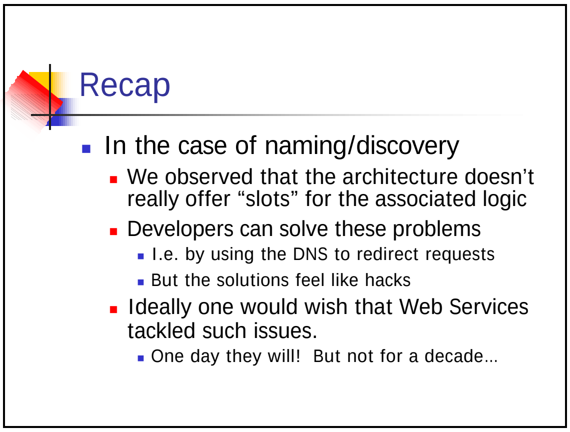
Recap

- We started by thinking about Web Services
  - Basically, a standardized architecture that clients client systems talk to servers
  - Uses XML and other Web protocols
  - And will be widely popular ("ubiquitous")
- Our goal is to build "trustworthy" systems using these standard, off-the-shelf techniques
- So we started to look at the issues top down



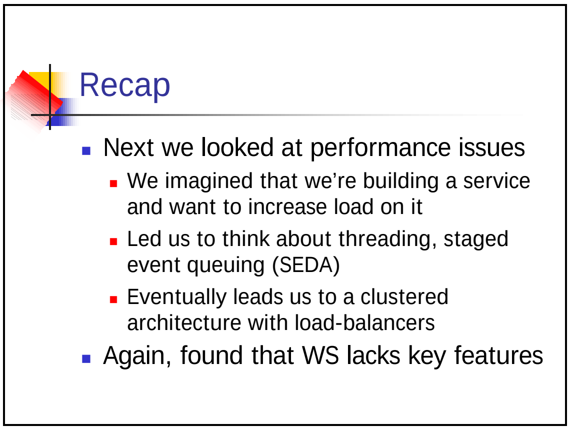
Recap

- First we looked at naming/discovery
  - We asked what decisions need to be made
    - Client needs to pick the right service
      - I want this particular database, or display device
    - Service may have a high-level routing decision
      - Send "East Coast" requests to the New Jersey center
    - Service also makes lower-level decisions
      - John Smith is doing a transaction; send requests to the same node if possible to benefit from caching
    - And finally the network does routing



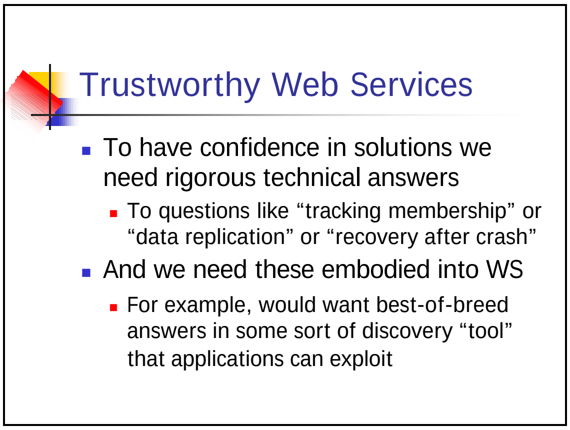
Recap

- In the case of naming/discovery
  - We observed that the architecture doesn't really offer "slots" for the associated logic
  - Developers can solve these problems
    - I.e. by using the DNS to redirect requests
    - But the solutions feel like hacks
  - Ideally one would wish that Web Services tackled such issues.
    - One day they will! But not for a decade…



Recap

- Next we looked at performance issues
  - We imagined that we're building a service and want to increase load on it
  - Led us to think about threading, staged event queuing (SEDA)
  - Eventually leads us to a clustered architecture with load-balancers
- Again, found that WS lacks key features



Trustworthy Web Services

- To have confidence in solutions we need rigorous technical answers
  - To questions like "tracking membership" or "data replication" or "recovery after crash"
- And we need these embodied into WS
  - For example, would want best-of-breed answers in some sort of discovery "tool" that applications can exploit

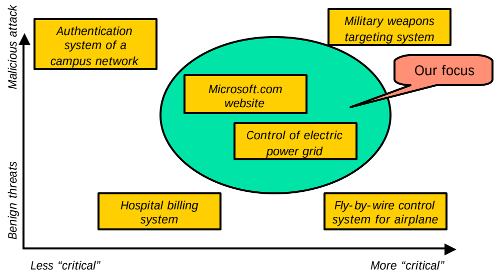
Trustworthy Computing

- Overall, we want to feel confident that the systems we build are "trustworthy"
- But what should this mean, and how realistic a goal is it?
- Today
  - Discuss some interpretations of the term
  - Settle on the "model" within which we'll work during the remainder of the term

Categories of systems…

- Roles computing systems play vary widely
  - Most computing systems aren't critical in a minute-by-minute sense
  - … but some systems matter more; if they are down, the enterprise is losing money
  - … and very rarely, we need to build ultra-reliable systems for mission-critical uses

Examples


Malicious attack
Benign threats
Authentication system of a campus network
Military weapons targeting system
Our focus
Microsoft.com website
Control of electric power grid
Hospital billing system
Fly-by-wire control system for airplane
Less "critical"
More "critical"

Techniques vary!

- Less critical systems that face accident (not attack) lend themselves to cheaper solutions
  - Particularly if we don't mind outages when something crashes
  - High or continuous availability is harder
- The mixture of *time-critical, very secure, very high availability* is particularly difficult
  - Solutions don't integrate well with standard tools
  - "Secure and highly available" can also be *slow*

Importance of "COTS"

- The term means "commercial off the shelf"
- To understand importance of COTS we need to understand history of computing
  - Prior to 1980, "roll your own" was common
  - But then with CORBA (and its predecessors) well-supported standards won the day
  - Productivity benefits of using standards are *enormous:* better development tools, better system management support, better feature sets
- Today, most projects mandate COTS

The dilemma

- But major products have been relaxed about:
  - Many aspects of security
  - Reliability
  - Time-critical computing (not the same as "fast")
- Jim Gray: "*Microsoft is mostly interested in multi-billion dollar markets. And it isn't feasible to make 100% of our customers happy. If we can make 80% of them happy 90% of the time, we're doing just fine.*"

Are COTS trustworthy?

- Security is improving but still pretty weak
  - Data is rarely protected “on the wire”
  - Systems are not designed with the threat of overt attack in mind
  - Often limited to perimeter security; if the attacker gets past the firewall, she’s home free
- Auditing and system management functions are frequently inadequate

Are COTS trustworthy?

- Most COTS technologies do anticipate crashes and the need to restart
  - You can usually ask the system to watch your application and relaunch after failure
  - You can even ask for a restart on a different node… but there won’t be any protection against split-brain problems
- So-called “transactional” model can help
  - Alternatively can make checkpoints, or replicate critical data, but without platform help

Is this enough?

- The way COTS systems provide restart is potentially slow
  - Transactional “model” can’t offer high availability (we’ll see why later)
  - Often must wait for failed machine to reboot, clean up its data structures, relaunch its main applications, etc
- In big commercial systems could be minutes or even hours
- Not enough… if we want high availability

Are COTS trustworthy?

- Security… reliability… what about:
  - Time-critical applications, where we want to guarantee a response within some bounded time (and know that the application is fast enough… but worry about platform overheads and delays)
  - Issues of system administration and management and upgrade

SoS and SOAs

- The trend is towards
  - Systems of Systems (SoS): federation of big existing technologies
  - Service Oriented Architectures (SOAs).
    - Object oriented or Web Services systems
    - Components declare their interfaces using an interface definition language (IDL) or a description language (WSDL)
    - Implementation is “hidden” from clients

Example: the Air Force JBI

Globally Interoperable Information “Space” that …

Aggregates, fuses, and disseminates tailored battlespace information to all echelons of a JTF

Decision-Quality Information

Links JTF sensors, systems & users together for unity of effort

Integrates legacy C2 resources

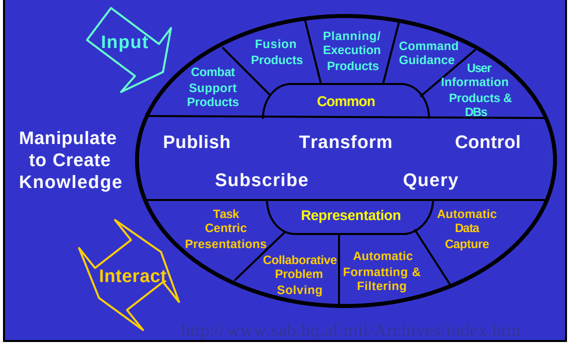
Focuses on Decision-Making

Enables Affordable Technology Refresh

Leverages Emerging Commercial Technologies

Inside the Battlespace InfoSphere

(circa 1999)


Input
Combat Support Products
Fusion Products
Planning/ Execution Products
Command Guidance
User Information Products & DBs
Common
Manipulate to Create Knowledge
Publish
Transform
Control
Subscribe
Query
Task Centric Presentations
Representation
Automatic Data Capture
Collaborative Problem Solving
Automatic Formatting & Filtering
Interact

http://www.sab.hq.af.mil/Archives/index.htm

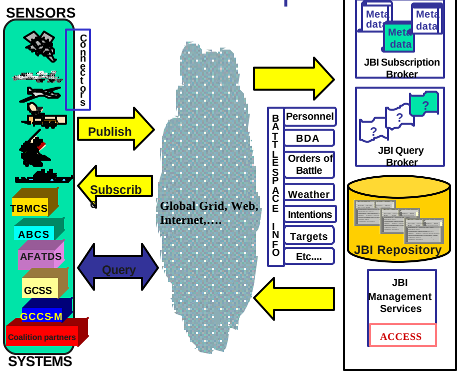
JBI Basics

The JBI is a system of systems that integrates, aggregates, & distributes information to users at all echelons, from the command center to the battlefield.

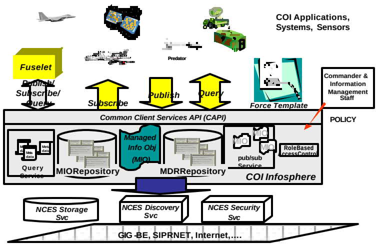
The JBI is built on four key technologies:

- Information exchange
  - **Publish / Subscribe / Query**
- Transforming data to knowledge
  - **Fuselets**
- Distributed collaboration
  - **Shared, updateable knowledge objects**
- Force/Unit interfaces
  - **Templates**
    - **Operational capability**
    - **Information inputs**
    - **Information requirements**

Architectural Concept



A fusion of BIG systems

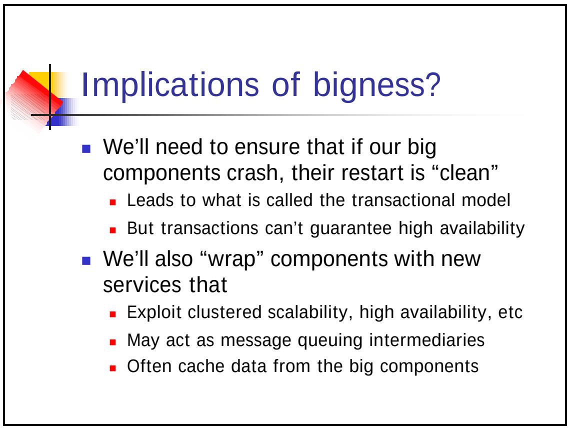


Observations?

- Everyone is starting to think big, not just the US Air Force
- Big systems are staggeringly complex
  - They won’t be easy to build
  - And will be even harder to operate and repair when problems occur
- Yet the payoff is huge and we often have no choice except to push forward!

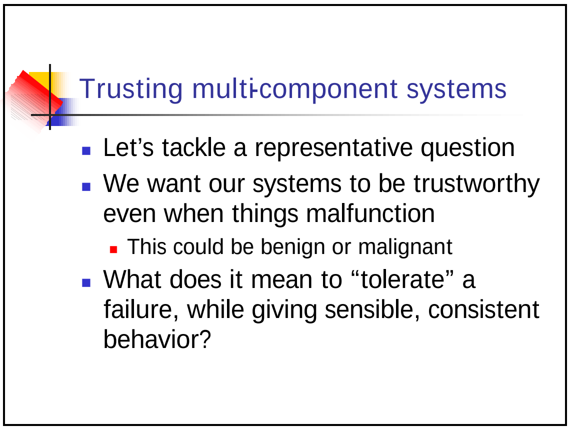
Systems of Systems (SoS) and Service Oriented Architectures (SOAs)

- The trends (towards huge systems) run against trustworthiness goals
  - IDL and WSDL specifications rarely include information about expected performance, security or reliability properties
  - And if they did… the platforms lack ways to enforce guarantees



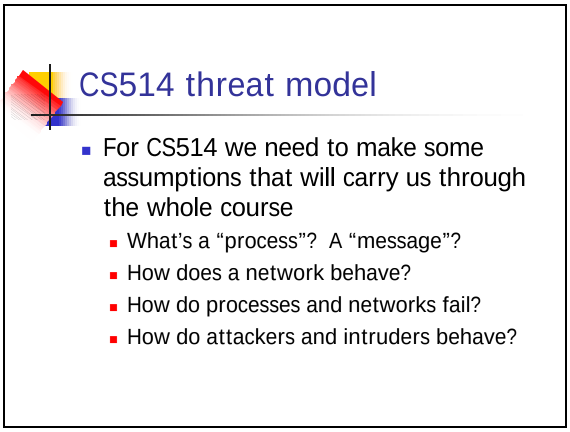
Implications of bigness?

- We'll need to ensure that if our big components crash, their restart is "clean"
  - Leads to what is called the transactional model
  - But transactions can't guarantee high availability
- We'll also "wrap" components with new services that
  - Exploit clustered scalability, high availability, etc
  - May act as message queuing intermediaries
  - Often cache data from the big components



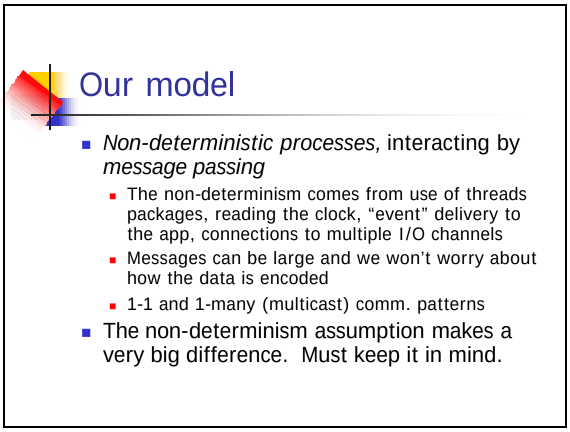
Trusting multi-component systems

- Let's tackle a representative question
- We want our systems to be trustworthy even when things malfunction
  - This could be benign or malignant
- What does it mean to "tolerate" a failure, while giving sensible, consistent behavior?



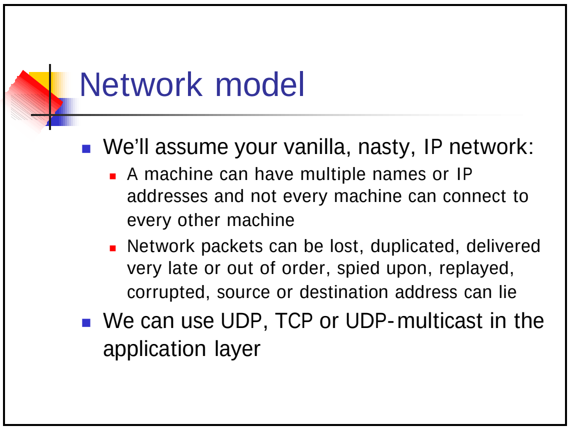
CS514 threat model

- For CS514 we need to make some assumptions that will carry us through the whole course
  - What's a "process"? A "message"?
  - How does a network behave?
  - How do processes and networks fail?
  - How do attackers and intruders behave?



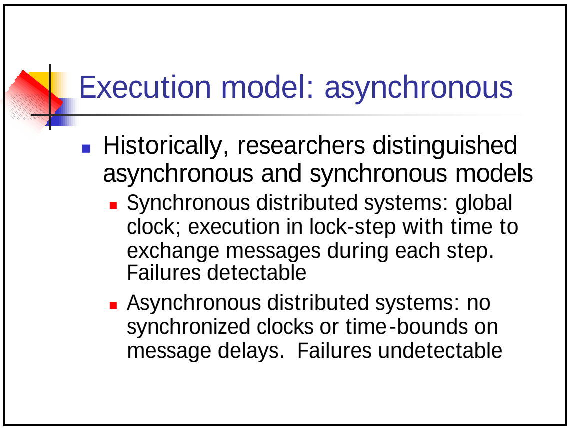
Our model

- *Non-deterministic processes,* interacting by *message passing*
  - The non-determinism comes from use of threads packages, reading the clock, "event" delivery to the app, connections to multiple I/O channels
  - Messages can be large and we won't worry about how the data is encoded
  - 1-1 and 1-many (multicast) comm. patterns
- The non-determinism assumption makes a very big difference. Must keep it in mind.



Network model

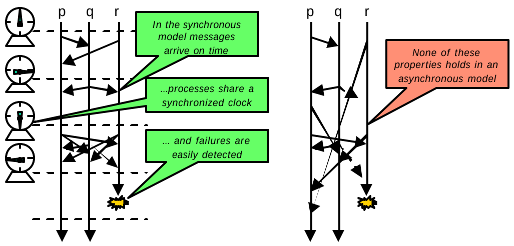
- We'll assume your vanilla, nasty, IP network:
  - A machine can have multiple names or IP addresses and not every machine can connect to every other machine
  - Network packets can be lost, duplicated, delivered very late or out of order, spied upon, replayed, corrupted, source or destination address can lie
- We can use UDP, TCP or UDP-multicast in the application layer



Execution model: asynchronous

- Historically, researchers distinguished asynchronous and synchronous models
  - Synchronous distributed systems: global clock; execution in lock-step with time to exchange messages during each step. Failures detectable
  - Asynchronous distributed systems: no synchronized clocks or time-bounds on message delays. Failures undetectable

Synchronous and Asynchronous Executions


p
q
r
In the synchronous model messages arrive on time
...processes share a synchronized clock
... and failures are easily detected
p
q
r
None of these properties holds in an asynchronous model

Reality: neither one

- Real distributed systems aren't synchronous
  - Although a flight control computer can come close
- Nor are they asynchronous
  - Software often treats them as asynchronous
  - In reality, clocks work well... so in practice we often use time cautiously and can even put limits on message delays
- For our purposes we usually start with an asynchronous model
  - Subsequently enrich it with sources of time when useful.
  - We sometimes assume a "public key" system. This lets us sign or encrypt data where need arises

Failure model

- How do real systems fail?
  - Bugs in applications are a big source of crashes. Often associated with non-determinism, which makes debugging hard
  - Software or hardware failures that crash the whole computer are also common
  - Network outages cause spikes of high packet loss or complete disconnection
  - Overload is a surprisingly important risk, too

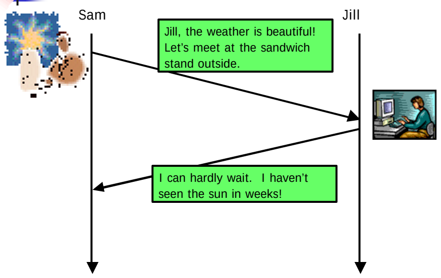
Detecting failures

- This can be hard!
  - An unresponsive machine might be working but temporarily "partitioned" away
  - A faulty program may continue to respond to some kinds of requests (it just gives incorrect responses)
  - Timeouts can be triggered by overloads
  - One core problem can cascade to trigger many others
- We usually know when things are working but rarely know what went wrong

Thought problem

- Jill and Sam will meet for lunch. They'll eat in the cafeteria unless both are sure that the weather is good
  - Jill's cubicle is inside, so Sam will send email
  - Both have lots of meetings, and might not read email. So she'll acknowledge his message.
  - They'll meet inside if one or the other is away from their desk and misses the email.
- Sam sees sun. Sends email. Jill acks's. Can they meet outside?

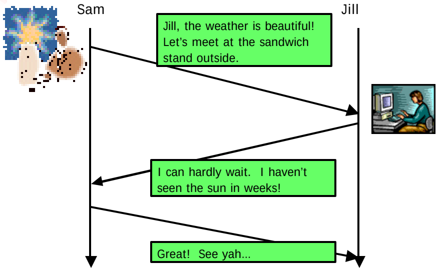
Sam and Jill


Sam
Jill
Jill, the weather is beautiful! Let's meet at the sandwich stand outside.
I can hardly wait. I haven't seen the sun in weeks!

They eat *inside!* Sam reasons:

- “Jill sent an acknowledgement but doesn’t know if I read it
- “If I didn’t get her acknowledgement I’ll assume she didn’t get my email
- “In that case I’ll go to the cafeteria
- “She’s uncertain, so she’ll meet me there

Sam had better send an Ack



Jill, the weather is beautiful! Let’s meet at the sandwich stand outside.

I can hardly wait. I haven’t seen the sun in weeks!

Great! See yah...

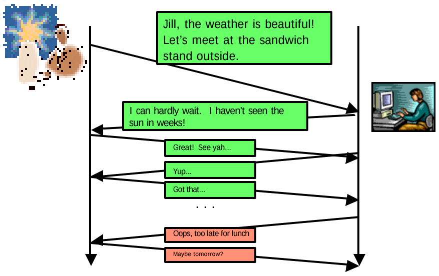
Why didn’t this help?

- Jill got the ack... but she realizes that Sam won’t be sure she got it
- Being unsure, he’s in the same state as before
- So he’ll go to the cafeteria, being dull and logical. And so she meets him there.

New and improved protocol

- Jill sends an ack. Sam acks the ack. Jill acks the ack of the ack....
- Suppose that noon arrives and Jill has sent her 117’th ack.
  - Should she assume that lunch is outside in the sun, or inside in the cafeteria?

How Sam and Jill’s romance ended



Jill, the weather is beautiful! Let’s meet at the sandwich stand outside.

I can hardly wait. I haven’t seen the sun in weeks!

Great! See yah...

Yup...

Got that...

. . .

Oops, too late for lunch

Maybe tomorrow?

Things we just can’t do

- We can’t detect failures in a trustworthy, consistent manner
- We can’t reach a state of “common knowledge” concerning something not agreed upon in the first place
- We can’t guarantee agreement on things (election of a leader, update to a replicated variable) in a way certain to tolerate failures

Consistency

- At the core of the notion of trust is a fundamental concept: “distributed consistency”
  - Our SoS has multiple components
  - Yet they behave as a single system: many components mimic a single one
- Examples:
  - Replicating data in a primary -backup server
  - Collection of clients agreeing on which to use
  - Jill and Sam agreeing on where to meet for lunch

Does this matter in big systems?

- Where were Jill and Sam in the JBI?
  - Well, JBI is supposed to coordinate military tacticians and fighters…
    - Jill and Sam are trying to coordinate too.
    - If they can’t solve a problem, how can the JBI?
    - Illustrates value of looking at questions in abstracted form!
  - Generalize: our big system can only solve “solvable” consistency problems!

Why is this important?

- Trustworthy systems, at their core, behave in a “consistent” way even when disrupted by failures, other stress
- Hence to achieve our goals we need to ask what the best we can do might be
  - If we set an impossible goal, we’ll fail!
  - But if we ignore consistency, we’ll also fail!

A bad news story?

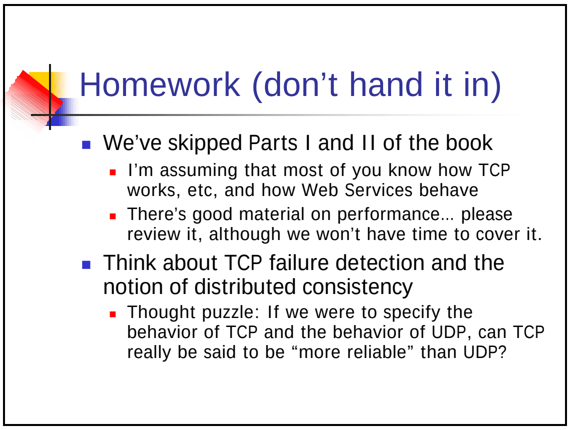
- Jill and Sam set out to solve an impossible problem
  - So for this story, yes, bad news
- Fortunately, there <u>are</u> practical options
  - If we pose goals carefully, stay out of trouble
  - Then solve problems and prove solutions correct!
- And insights from “small worlds” can often be applied to very big systems of systems

Trust and Consistency

- To be trustworthy, a system must provide guarantees and enforce rules
- When this entails actions at multiple places (or, equivalently, updating replicated data) we require consistency
- If a mechanism ensures that an observer can’t distinguish the distributed system from a non-distributed one, we’ll say it behaves consistently

Looking ahead

- We’ll start from the ground and work our way up, building a notion of consistency
  - First, consistency about temporal words like “A happened before B”, or “When A happened, process P believed that Q…”
  - Then we’ll look at a simple application of this to checkpoint/rollback
  - And then we’ll work up to a full-fledged mechanism for replicating data and coordinating actions in a big system



Homework (don't hand it in)

- We've skipped Parts I and II of the book
  - I'm assuming that most of you know how TCP works, etc, and how Web Services behave
  - There's good material on performance… please review it, although we won't have time to cover it.
- Think about TCP failure detection and the notion of distributed consistency
  - Thought puzzle: If we were to specify the behavior of TCP and the behavior of UDP, can TCP really be said to be "more reliable" than UDP?