Dependability, Abstraction, and Programming

David Lomet
Microsoft Research
lomet@microsoft.com

Dependability

- Randell:
  - *Dependability is the system property that integrates such attributes as availability, safety, security, survivability, maintainability.*
  - **Key point:** dependability is more than just availability
- Hoare: *
  - *The price of reliability is utter simplicity- and this is a price that major software manufacturers find too high to afford.*
  - **Key point:** unless it is easy, natural, and simple, programming for dependability may well compromise it.

  - * Tony now works for Microsoft

Current situation

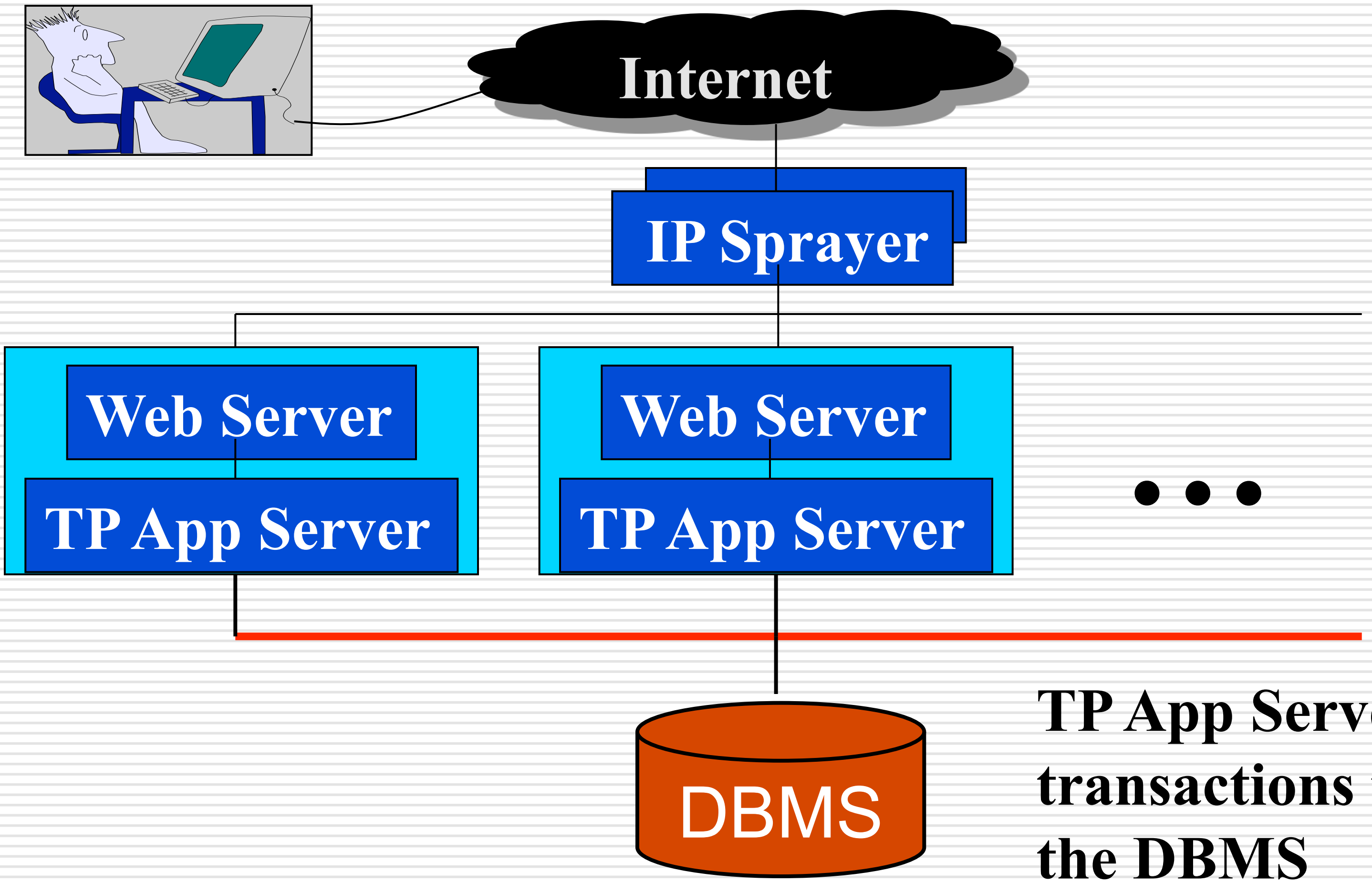
- Gray: *on availability*
  - *Everyone has a serious problem*
  - *The BEST people publish their stats*
  - *The others HIDE their stats*
  - **Key point:** we have a problem
- Patterson:
  - *Service outages are frequent*
    - *65% of IT managers report that their websites were unavailable over a 6-month period;*
  - *Outage costs are high*
    - *Social effects: negative press, loss of customers who click over to a competitor*
  - **Key point:** not only do we have a problem, but it is a costly problem

    - *Patterson argues for fast recovery*

Talk Outline

- **Dependability**
  - Need for dependability
  - Scalability and availability techniques
- Abstraction
  - What's right/wrong with transactions
  - State management in a "stateless" world
  - A new "abstraction (to be revealed)
- Making it work
  - Implementing the "new" abstraction
  - Phoenix project approach
  - "Magic" applied to problem
- Summary

Ex: An E-Commerce Server



TP App Server uses transactions to access the DBMS

Scalability/Availability Techniques

- Web based enterprise systems scale
  - Frequently with decent availability
- Key is “stateless” mid-tier servers
  - Application instantiated anywhere in middle tier
  - No difficulty re-instantiating elsewhere
- But there is state
  - Just not in the execution state
  - How to handle it??

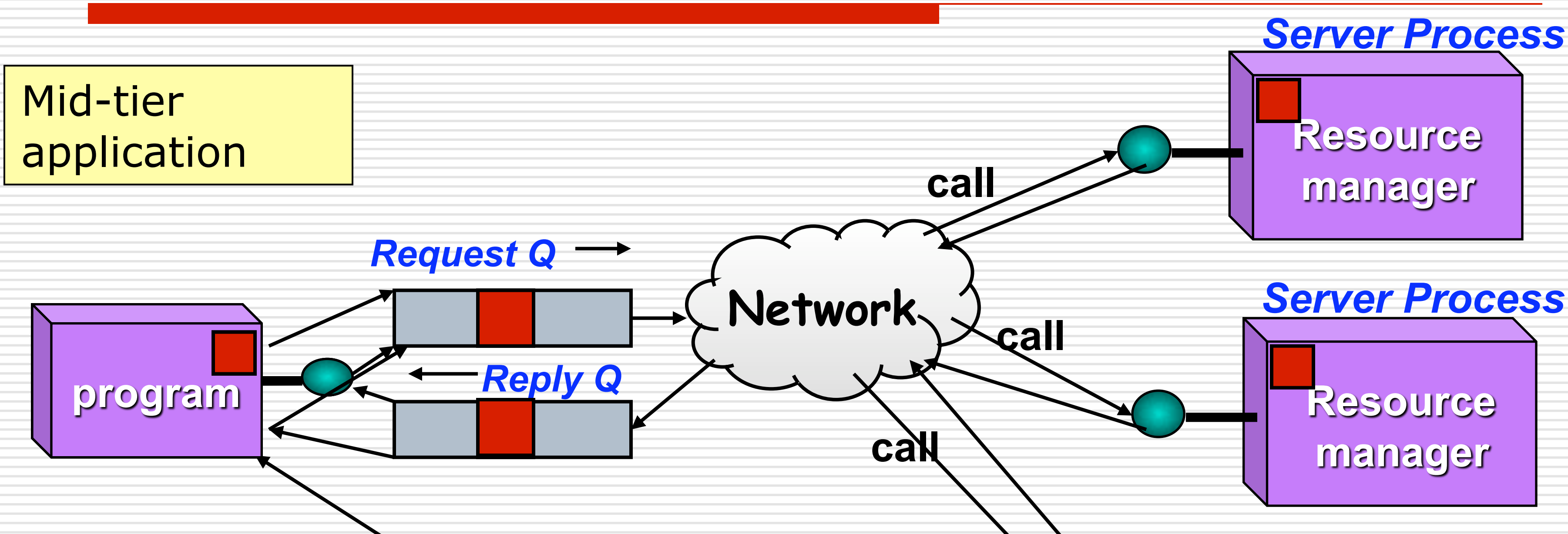
Talk Outline

- Dependability
  - Need for dependability
  - Scalability and availability techniques
- **Abstraction**
  - What’s right/wrong with transactions
  - State management in a “stateless” world
  - A new “abstraction (to be revealed)
- Making it work
  - Implementing the “new” abstraction
  - Phoenix project approach
  - “Magic” applied to problem
- Summary

Transactions are Terrific but…

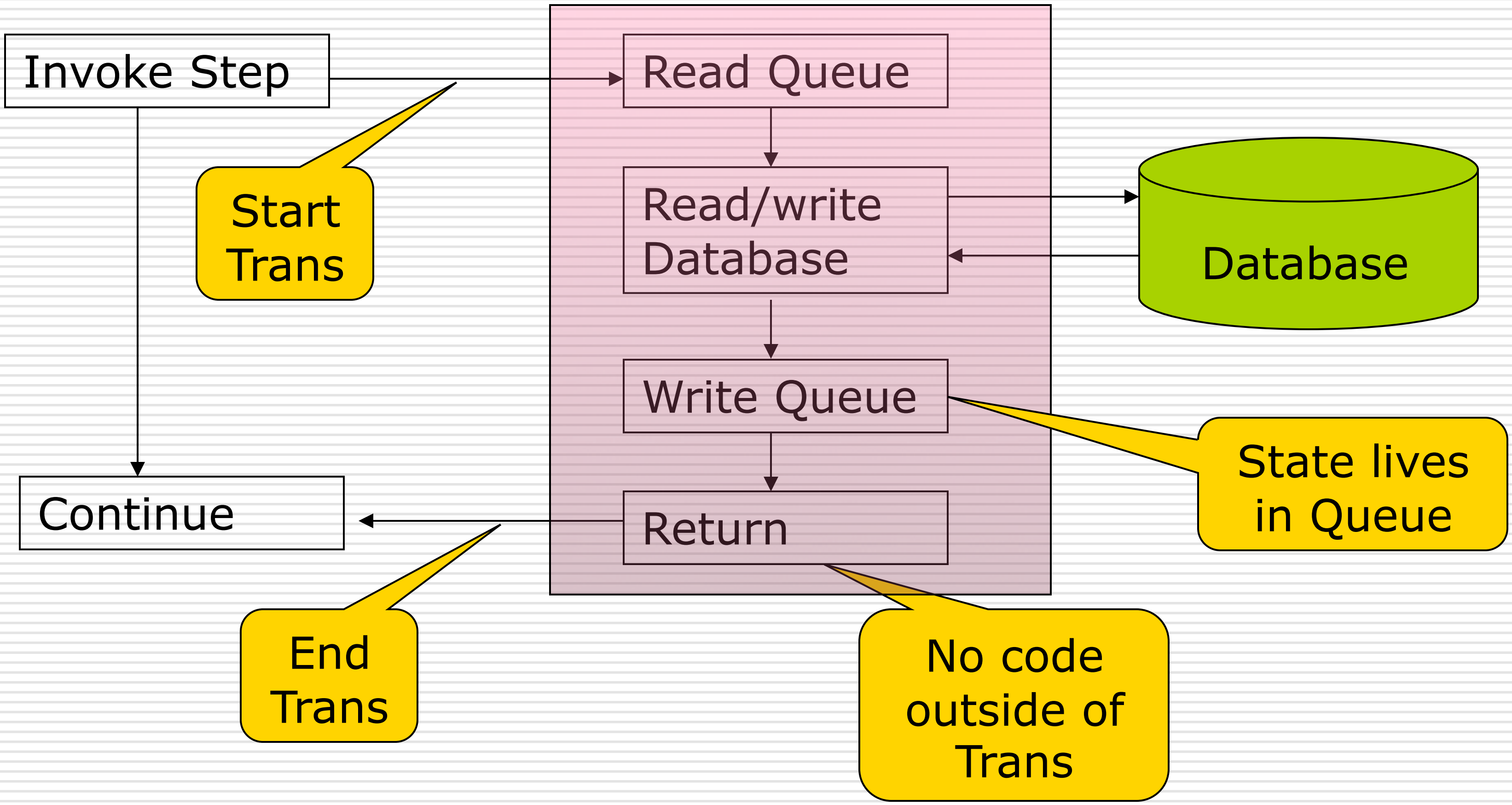
- Why terrific?
  - Clean semantics, simple and natural to use in database interactions, implementable with good performance
  - One of two abstractions upon which Database systems are built
    - Other is relational model
- What is the problem?
  - Half of enterprise system outside of transaction boundary- the application half
  - Databases recover to last transaction
  - What happens to applications?

Transactions for Applications



Each method call is a transaction
1. Read state from a transactional queue
2. Invoke the method
3. Write state to the transactional queue

Stateless Application Step



Problems

- **Two phase commit**
  - Performance and latency
  - Site autonomy (crucial in internet environment)
- **Error handling**
  - No part of program outside of transaction boundary
  - When app step is within a transaction, who handles transaction failures?
    - Not program logic-- At least not in middle tier
    - Frequently post to an error queue
- **Unnatural “string of beads” style**
  - Program needs to be re-arranged to fit model
  - Especially when multiple servers need to be involved
    - E.g. an airline and a car rental company
  - **Essentially, *programmer manages state***
  - Stored in database and/or transactional queue
  - Program organized to facilitate state management

We need a new “abstraction”

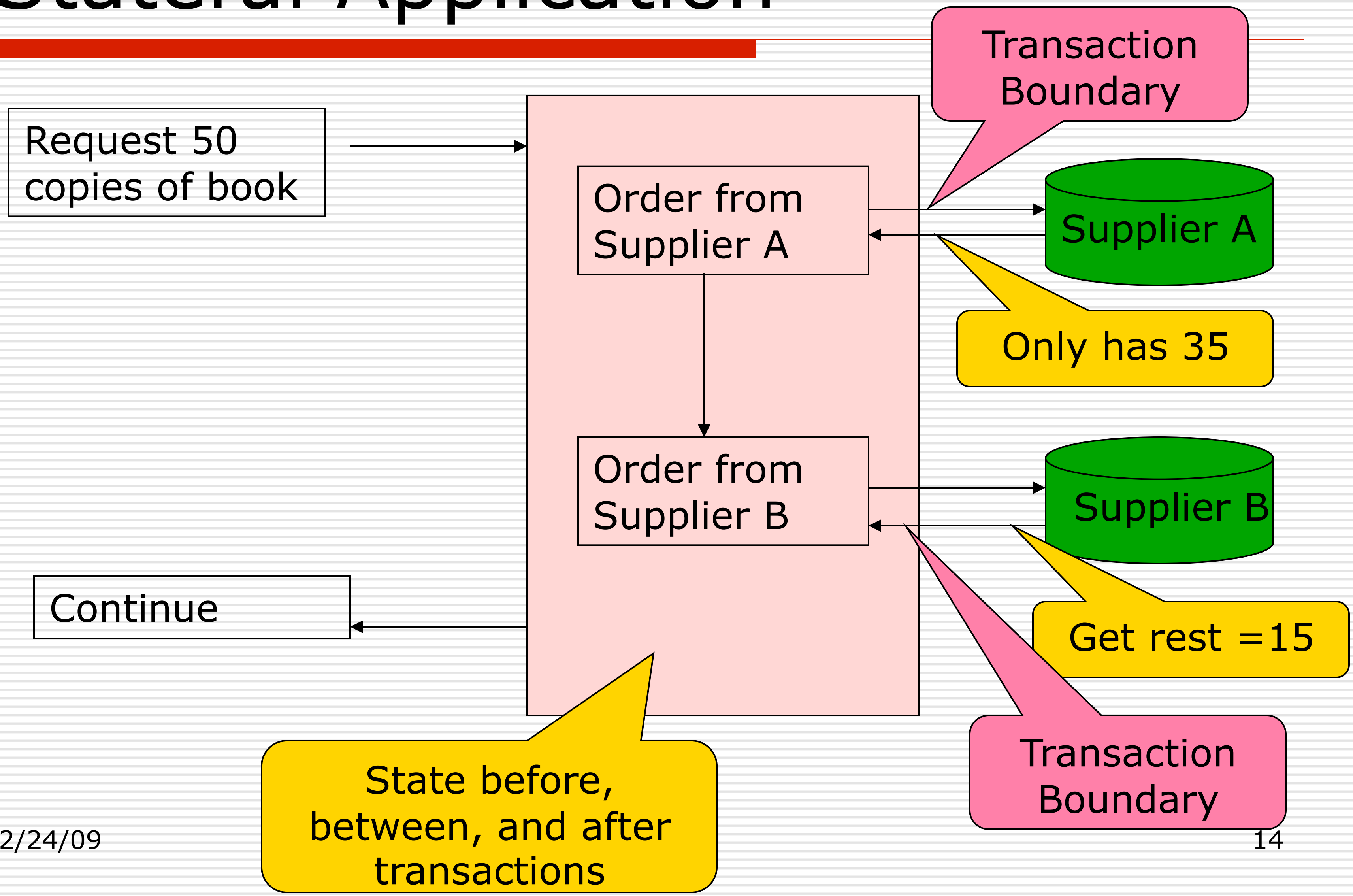
What's in a Good Abstraction

- Clean semantics, simple and natural
  - But this is not enough
  - "Do what I mean" is simple and natural
- Implementable with
  - Good performance
  - Robustness (reliable and predictable)
- SO THAT- programmer can delegate to the system some important aspect of his problem
  - If too complicated or not sufficiently robust, abstraction can get in the way
- Historical examples of great abstractions
  - **Transactions:** delegated concurrency control and recovery while presenting isolated view of system
  - **Relational model:** delegated physical database design and query processing/optimization to system while presenting "data independent" conceptual view

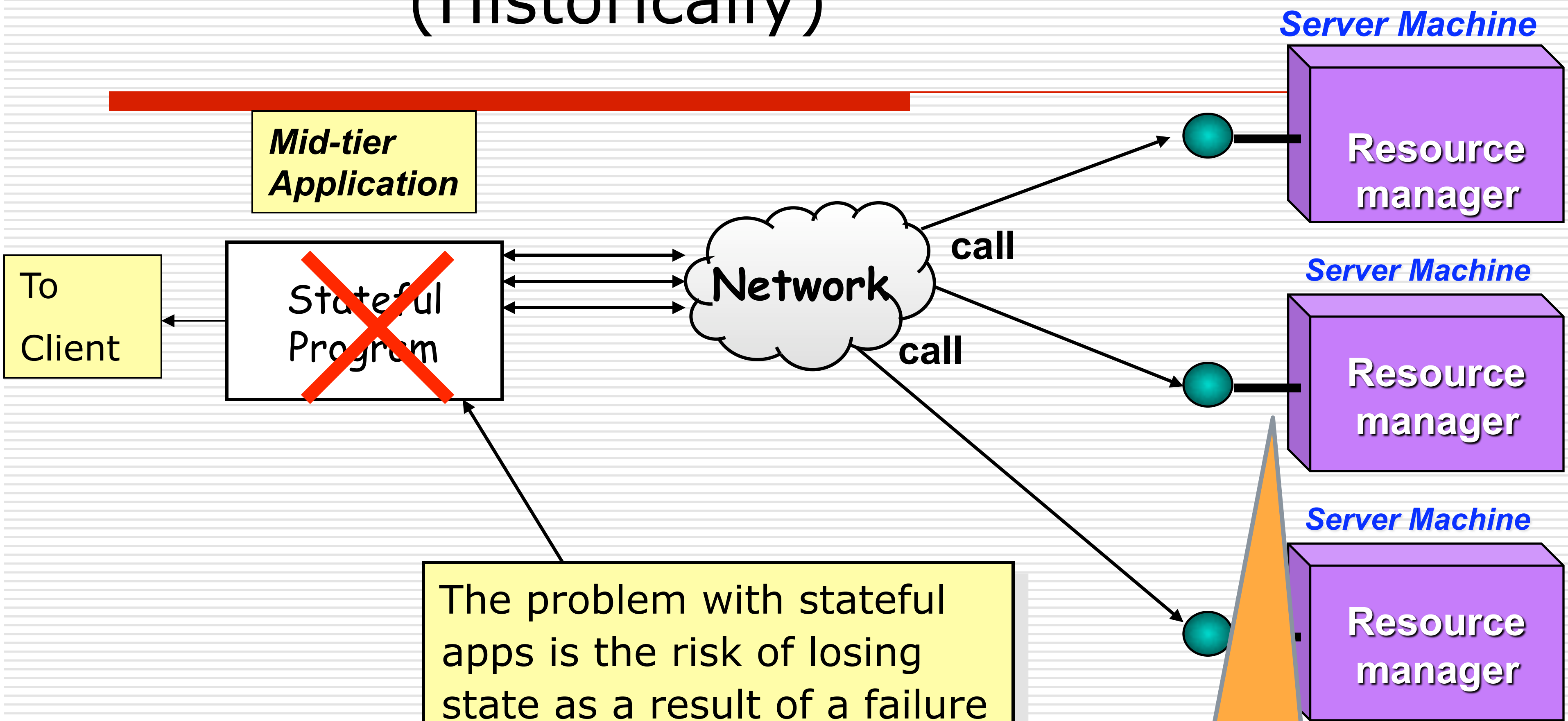
New Abstraction for Applications

- But old! **Stateful Programming Model**
  - Simple: programmers naturally do it!
  - Easy to understand
- Execution state captures much of the application state
  - Without having to otherwise manifest it
  - This part of state "manages itself" as program executes
    - Delegated to the system
  - Programmer can focus on "business logic"
    - Making program easier to write, understand, debug, maintain
- Must be applied to Enterprise Applications
  - Quest for "exactly once" execution semantics
    - Equivalent to failure-free execution
  - With high availability & scalability
  - Requires state persistence & management
- ***Well "Ha Ha!"***
  - Can't be made scalable & available!!!
  - How is state captured, moved, re-instantiated?

Stateful Application



Stateful Applications
(Historically)



The problem with stateful apps is the risk of losing state as a result of a failure

Any resource manager may have results of partial execution

Talk Outline

- Dependability
  - Need for dependability
  - Scalability and availability techniques
- Abstraction
  - What's right/wrong with transactions
  - State management in a "stateless" world
  - A new "abstraction (to be revealed)
- **Making it work**
  - Implementing the "new" abstraction
  - Phoenix project approach
  - "Magic" applied to problem
- Summary

Making Stateful Apps Work #1

- Transactions are expensive way to persist state
- Why not use REDO logging
    - Together with checkpointing
  - Old technology– now applied to enterprise computing
    - Used previously to avoid re-running of loooong apps
      - Frequently in context of scientific apps
  - Redo log captures non-determinism
    - Replay application from redo log of events
- Requires "pessimistic logging"
  - Log force whenever state is revealed to other parts of system (commits state)
  - We focus on optimizing pessimistic logging
    - Many log forces eliminated– published some *slick* methods
- Thus: system manages state by capturing execution state

Making Stateful Apps Work #2

- Providing Availability & Scalability
  - State is on log
    - So app can be deleted and re-instantiated for scalability, and can survive crashes
  - Application replayed from log to re-instantiate
    - After failure
    - Or for scalability, manageability
  - To redeploy elsewhere
    - Ship log elsewhere

Phoenix Project (with Roger Barga)

- Provide robust, dependable applications
  - Available, scalable, …
  - Using stateful program abstraction
    - No explicit program logic for dependability
    - That is delegated to Phoenix system
- Based on .NET infrastructure
  - Component software
  - Uses .NET remoting
    - Contexts, interception
    - Automatic logging of messages
  - Checkpointing, replay, monitoring, etc.

Phoenix Component Types

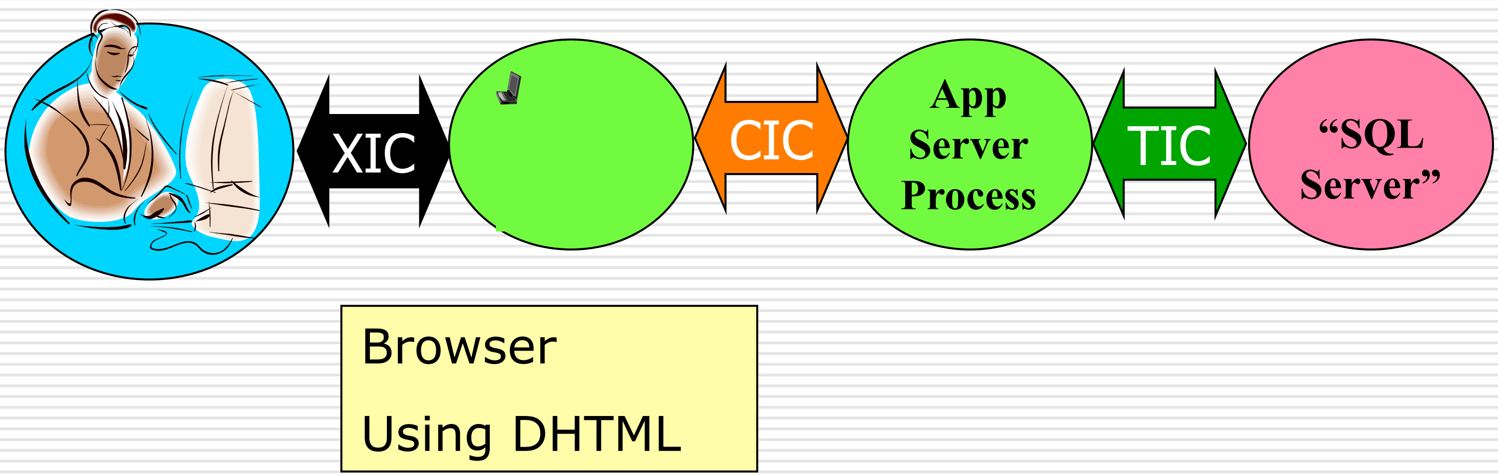
- Persistent(PCOM): e.g. web/application servers
  - This is stateful programming abstraction
  - State survives system failures
    - Via logging interactions with other system components
- Transactional(TCOM): e.g. SQL DB sessions
  - Transactions can abort- aborted state may "die"
    - But **committed** states, messages survive system failures
- External(XCOM): e.g. users, autonomous sites,etc.
  - Not under our control (*need do nothing*)
  - Prompt logging by PCOM limits exposure
  - Only failure during interaction is "unmasked"
    - E.g. user may have to re-enter data, see duplicate message

Interaction Contracts
Bi-lateral (sender/receiver) contracts

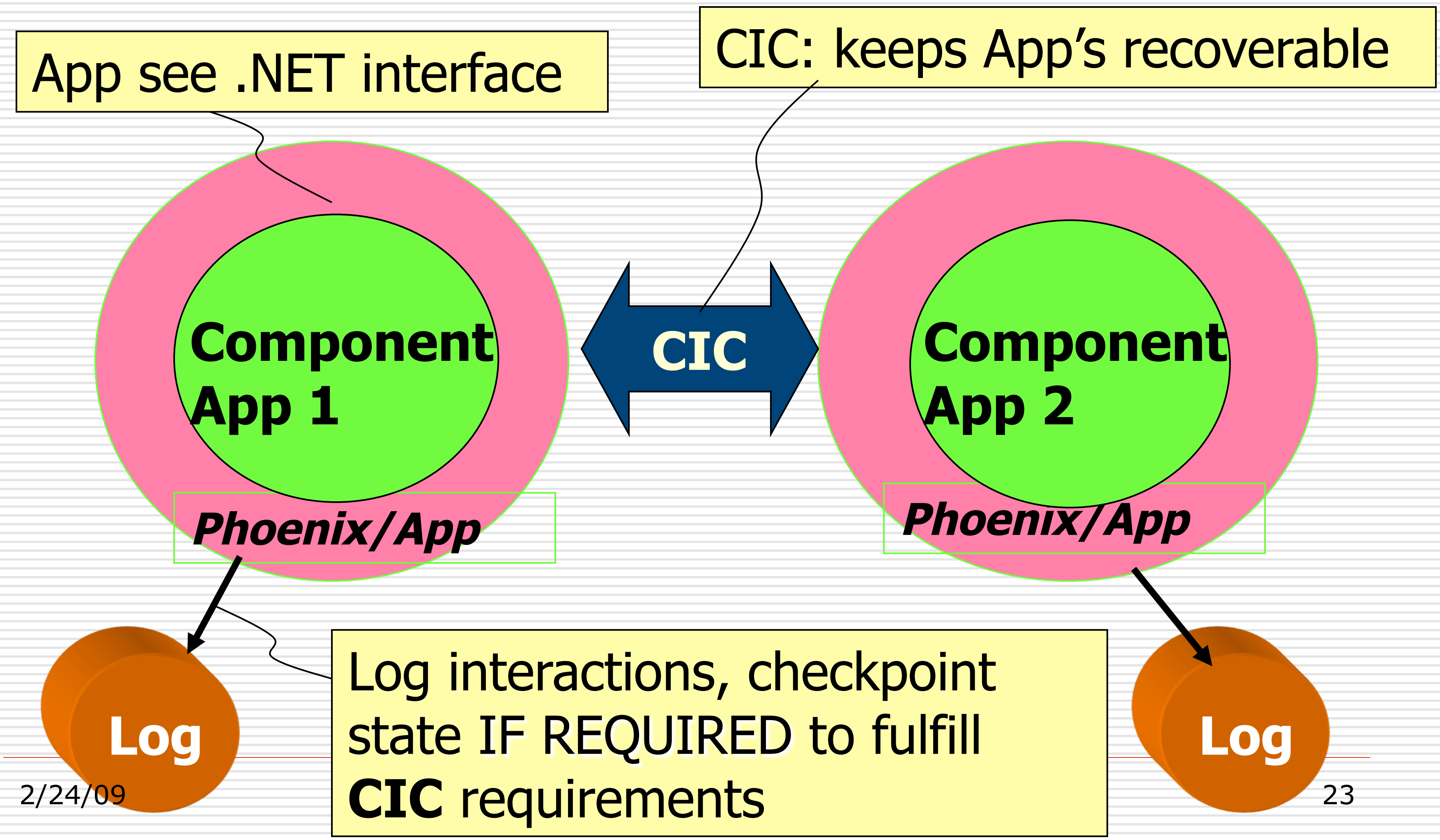
- **Committed interaction contract** (**CIC**)
  - ***PCOM⇔PCOM***
  - Guarantees that interaction persists across failures
- **Transactional interaction contract** (**TIC**)
  - ***PCOM⇔TCOM***
  - Permits transactional component to abort
  - But final commit is persistent
- **External interaction contract** (**XIC**)
  - ***PCOM⇔XCOM***
  - Permits interaction with external world, which does not play by our rules
  - Only failures during interaction are not masked

System Schematic

- Forms of Components and Interaction Contracts
  - **Persistent [PCOM]-** pervasive within system
  - **External [XCOM]-** at system edges (can initiate work)
  - **Transactional [TCOM]-** at system edges (receives work)



Recovery Infrastructure



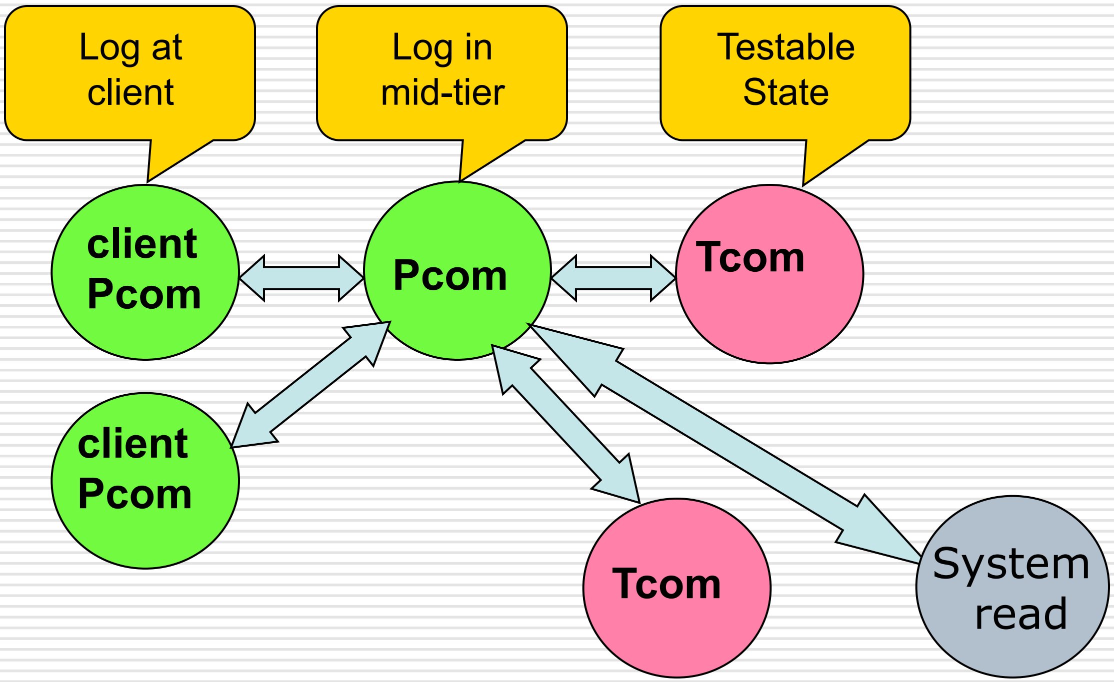
GREAT! But….

- To move application to another site
  - Requires shipping the log
  - Some current apps can be deployed anywhere
    - Because they are stateless
    - State captured at backend db or in cookie
    - These are limited and unnatural, but excellent in robustness
- So- can we have our cake and eat it too???
  - Do you need a hint about answer?

Magic….

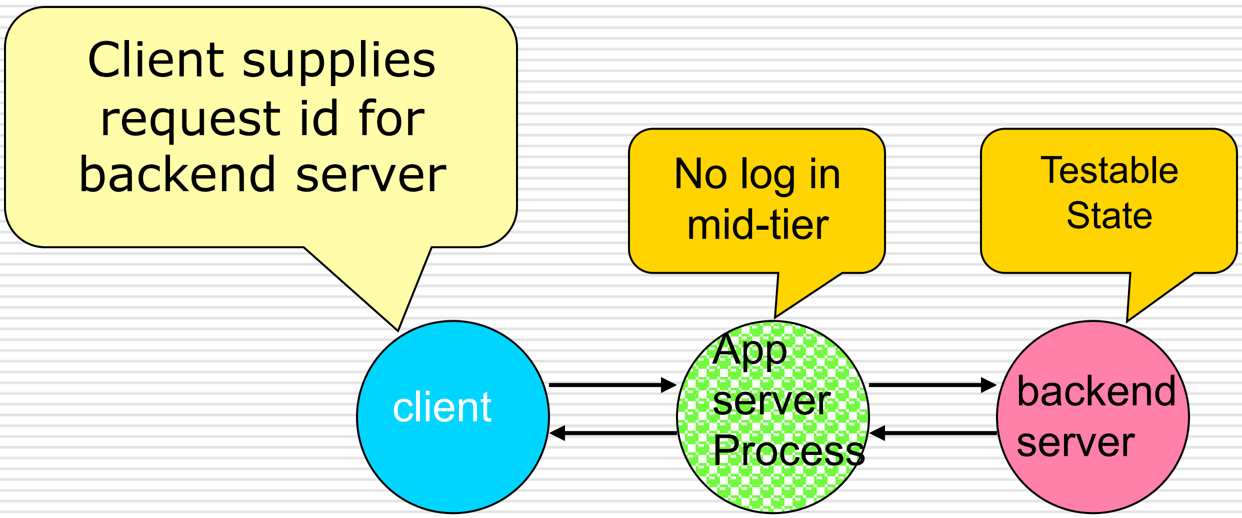
- *How is it possible to have persistent stateful applications without logging?*
- It isn't, but…
  - We can remove logging requirements from part of the system
    - Especially the middle tier
    - Permitting those apps to be failed over and re-instantiated anywhere
- There is a limited precursor for this
  - Though we will permit much more

Current Phoenix Model



Multiple clients have multiple message interactions with a mid-tier Pcom that does logging. The mid-tier Pcom interacts with multiple backend servers in multiple transactions, and may read state at any time.

e-Transactions *

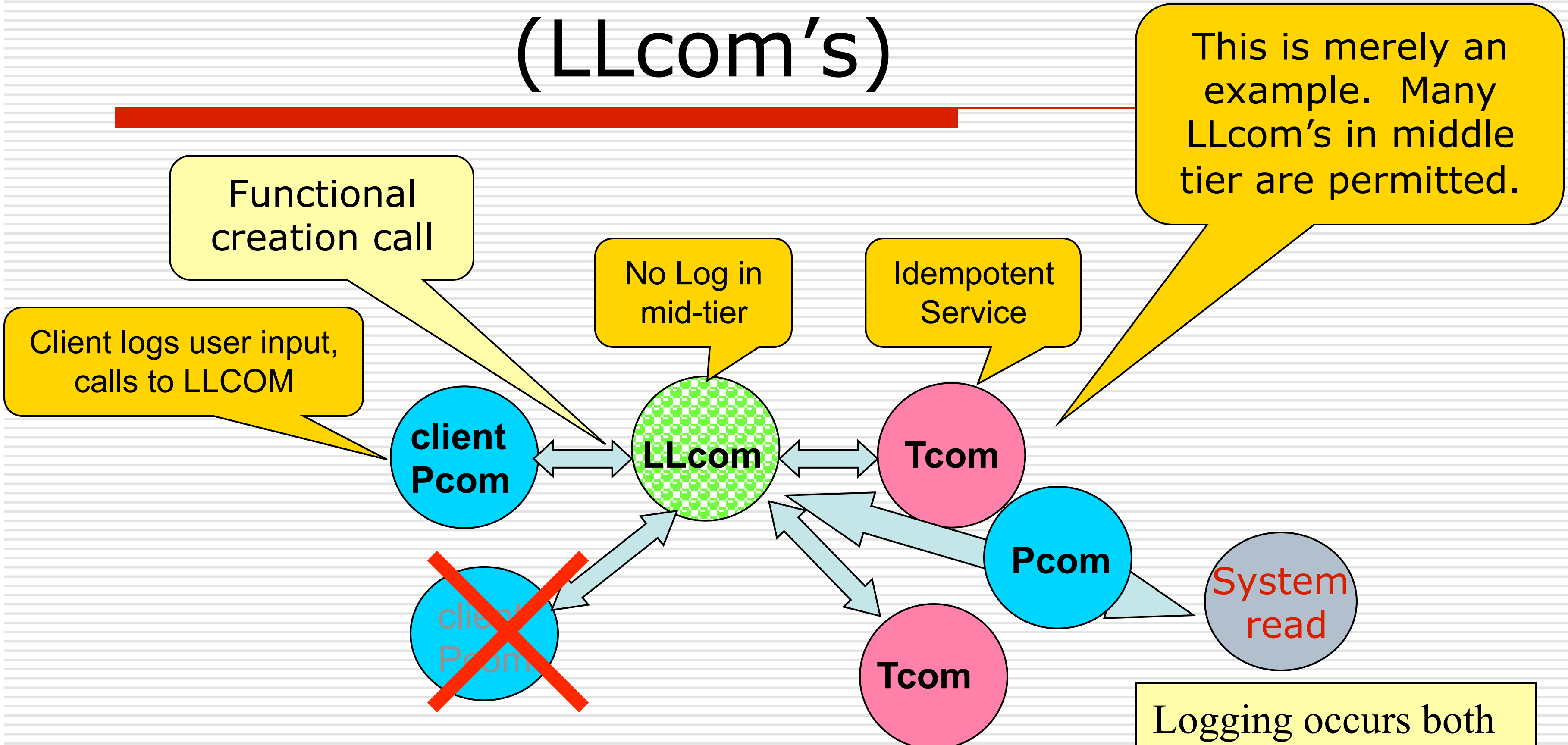


This is indeed a stateful application. State lives outside of backend transaction.

A single client has a single request/reply interaction with a middle-tier app server that does no logging. The app server interacts with the backend server in a single transaction, propagating client request id.

* S. Frolund and R. Guerraoui: A Pragmatic Implementation of e-Transactions. *19th IEEE Symp on Reliable Distributed Systems*, 186-195, (2000)

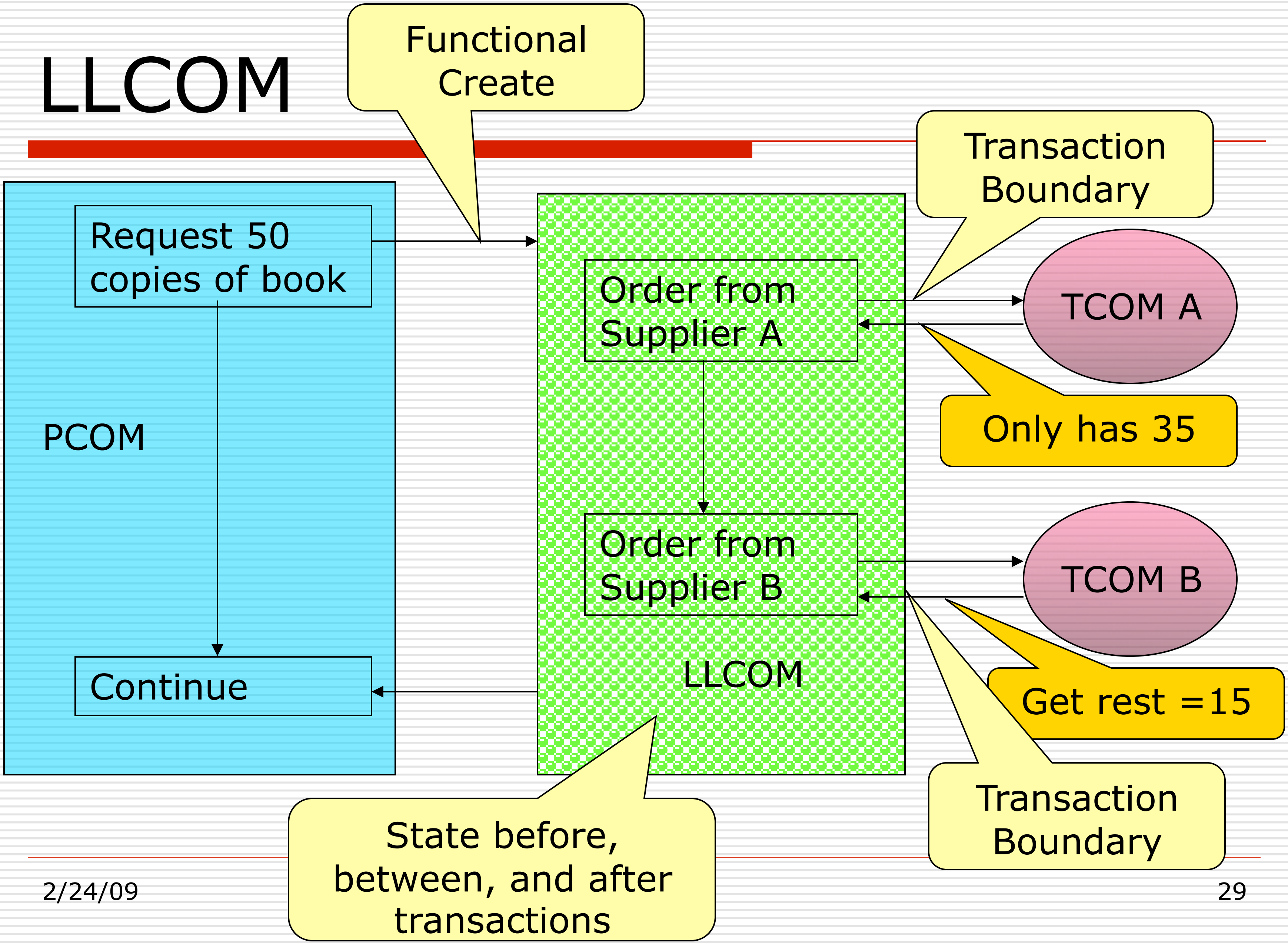
Logless Persistent Components (LLcom's)



Logging occurs both at client (Pcom) and at backend (Tcom) for testable state

A client can have multiple message interactions with a mid-tier **LLcom** that does not log. The LLcom interacts with multiple Tcoms- via **Idempotent** interactions.

LLCOM



Functional Create
Transaction Boundary
Request 50 copies of book
PCOM
Continue
Order from Supplier A
Order from Supplier B
LLCOM
TCOM A
TCOM B
Only has 35
Get rest =15
Transaction Boundary
State before, between, and after transactions

Some Real Pluses

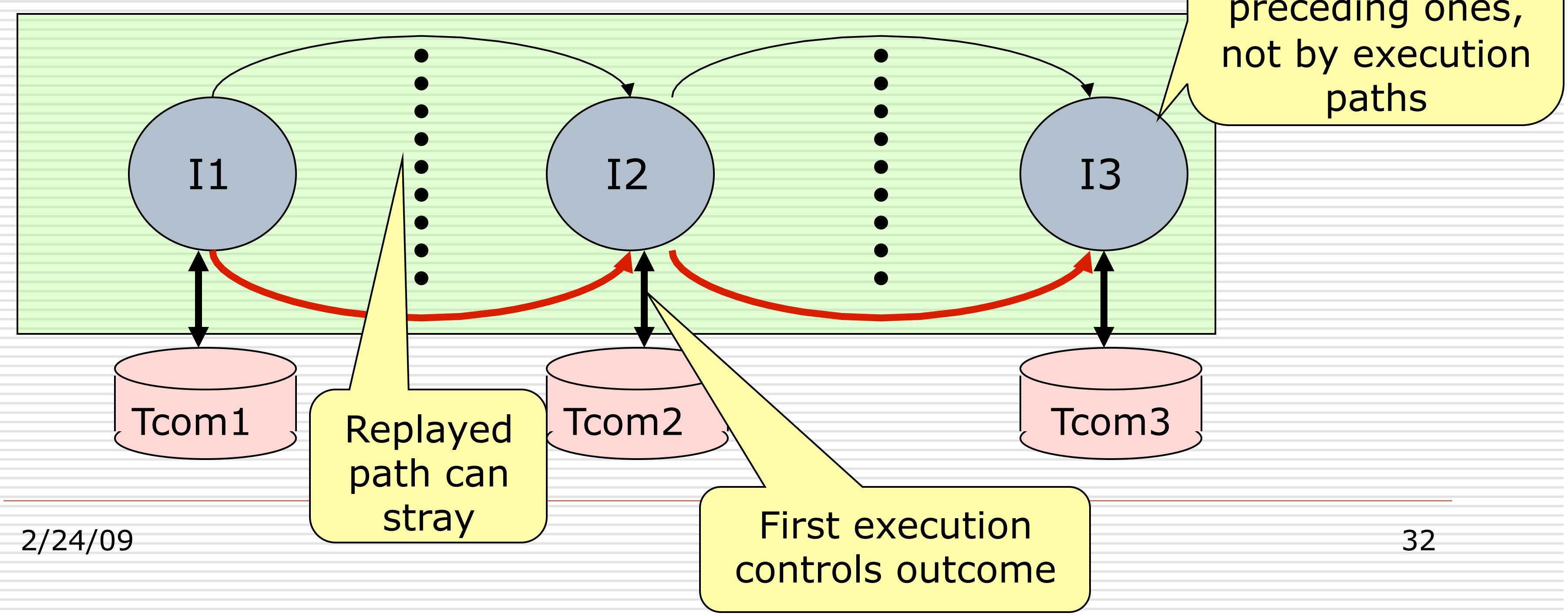
1. **No log vs optimized log**
   - No mechanism to support
2. **Middle tier not "recovery aware"**
   - Except to support "functional create"
3. **Excellent normal case performance**
   - No logging- indeed, no interception!
4. **No state needing to be shipped**
     - For failover, scalability, manageability
   - Create call simply goes to another system
5. **No increased logging elsewhere**
   - Tcom's must log for idempotence
     - In any event to cope with in-doubt outcomes
     - But will need to retain logged info longer
   - Pcom at client must log
     - In any event to capture user input

But there are limitations

- Middle tier cannot look around in deciding what to do in two ways
  - Cannot decide which deal to accept at a backend server based on reads
  - Cannot decide which backend server to invoke based on reads
- Fundamental problem is that reads are rarely idempotent
  - Can change on re-execution!
- We want to exploit non-idempotent reads
  - And still be recoverable

Mid-tier Faithless Replay

- **Goal: exactly-once at backend & client**
  - Mid-tier component replay can
    - Stray non-deterministically
    - So long as backend effect is determisitic



Enabling Non-idempotent Reads #1

Generalized Idempotent Request: GIR

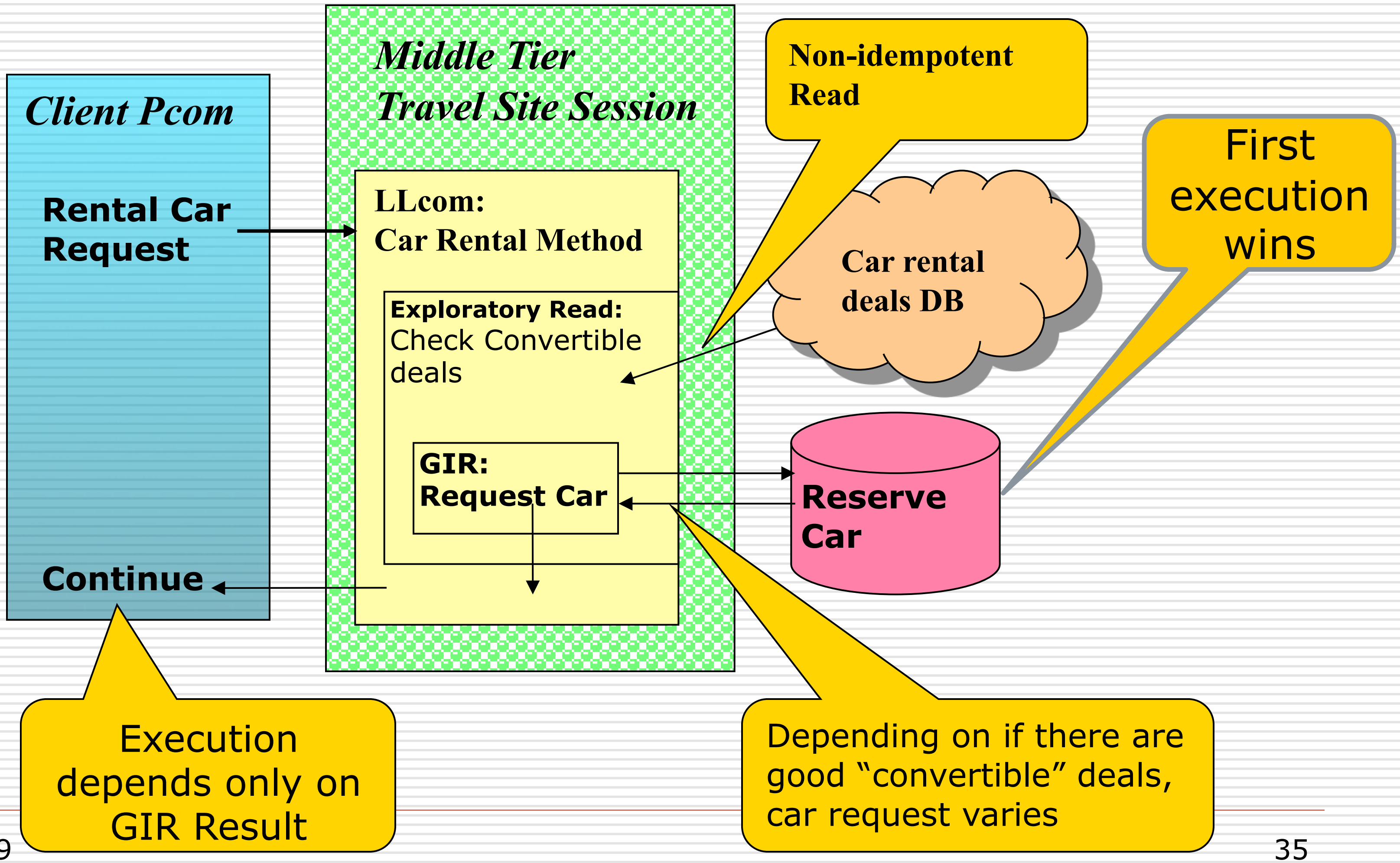
- **Idempotence:** duplicate requests are executed exactly once and return same reply
  - $IR(A_1,I_1) \circ IR(A_1,I_1) = IR(A_1,I_1)$
- **Generalized Idempotence:** requests with the same request id executed exactly once and return same reply
  - Even when other arguments are different!
  - Request ID's are normal message duplicate detection technique currently
  - $\mathbf{GIR(A_x,I_1)} \circ GIR(A_1,I_1) = GIR(A_1,I_1)$

Non-Idempotent Reads #2

LLcom restrictions

- **E-Proc:** for "exploratory" reads
  - Non-idempotent reads must occur only in E-proc
  - E-proc ends always with GIR request
    - To same service
    - With same request id
    - But potentially different other arguments
  - No posting of exploratory read info outside of E-proc
    - Only result of GIR request passed outward
- Only E-proc's GIR result impacts LLcom
  - E-proc is idempotent and
  - LLcom execution outside of E-proc is replayable

Example Application
"I want a convertible… maybe!"



Abort as “Exploratory Read”

- Put Abort inside E-proc
  - Multiple aborts prior to final commit
- Commit is GIR request
  - Transaction request args can differ
    - On each attempt to end transaction
    - So long as ID is the same
- E-proc exit after GIR Commit
- Need to permit Abort as GIR request
  - For Abort to be the final outcome
  - Permit program to request GIR abort

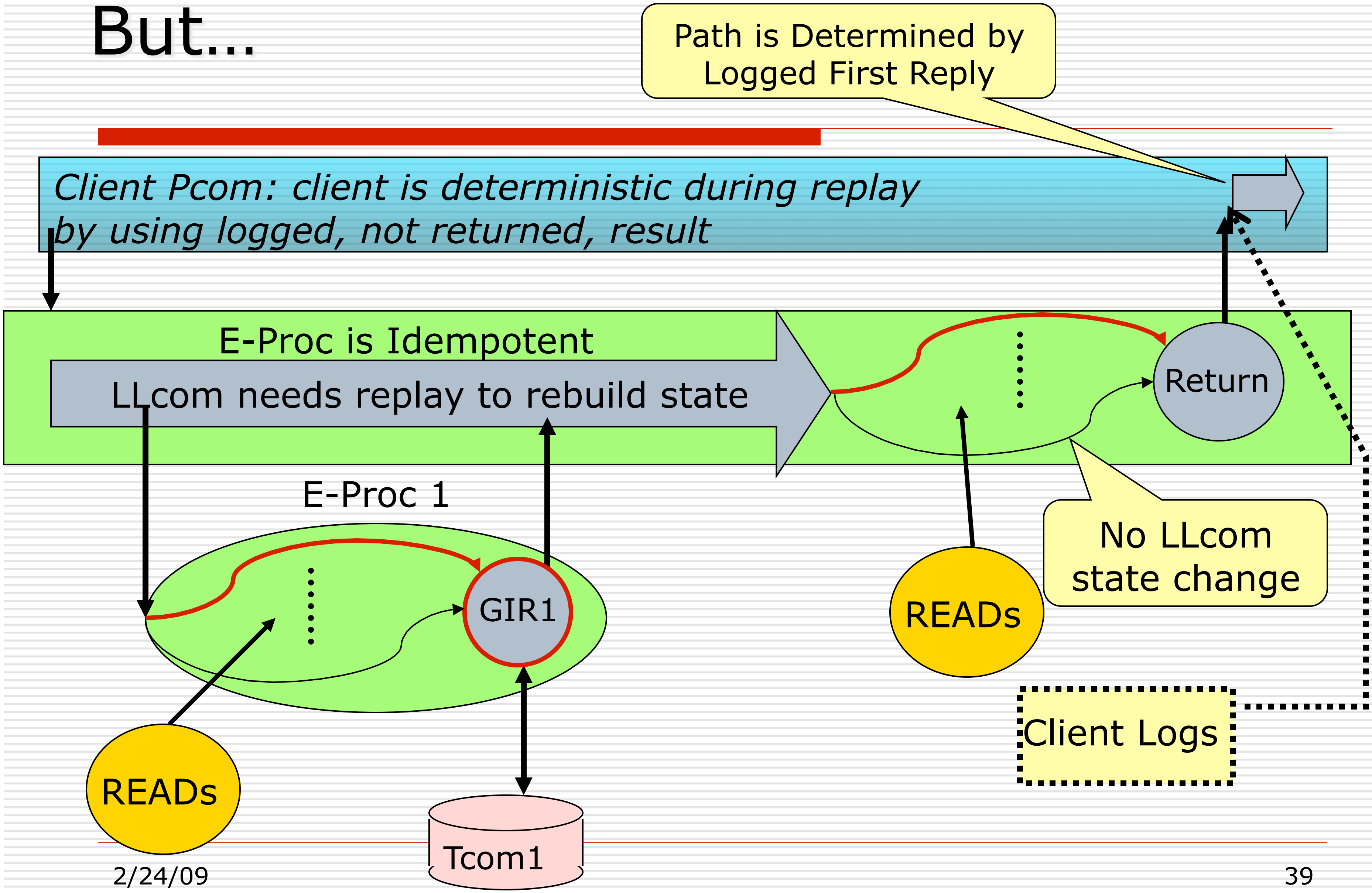
What's the GENERAL RULE?

- **Log after non-determinism**
  - Non-determinism during replay is ignored
  - Logging information determines subsequent execution
- **RESULT: FIRST EXECUTION WINS!**
  - This suggests exploiting…
  - Client logging as well as server logging

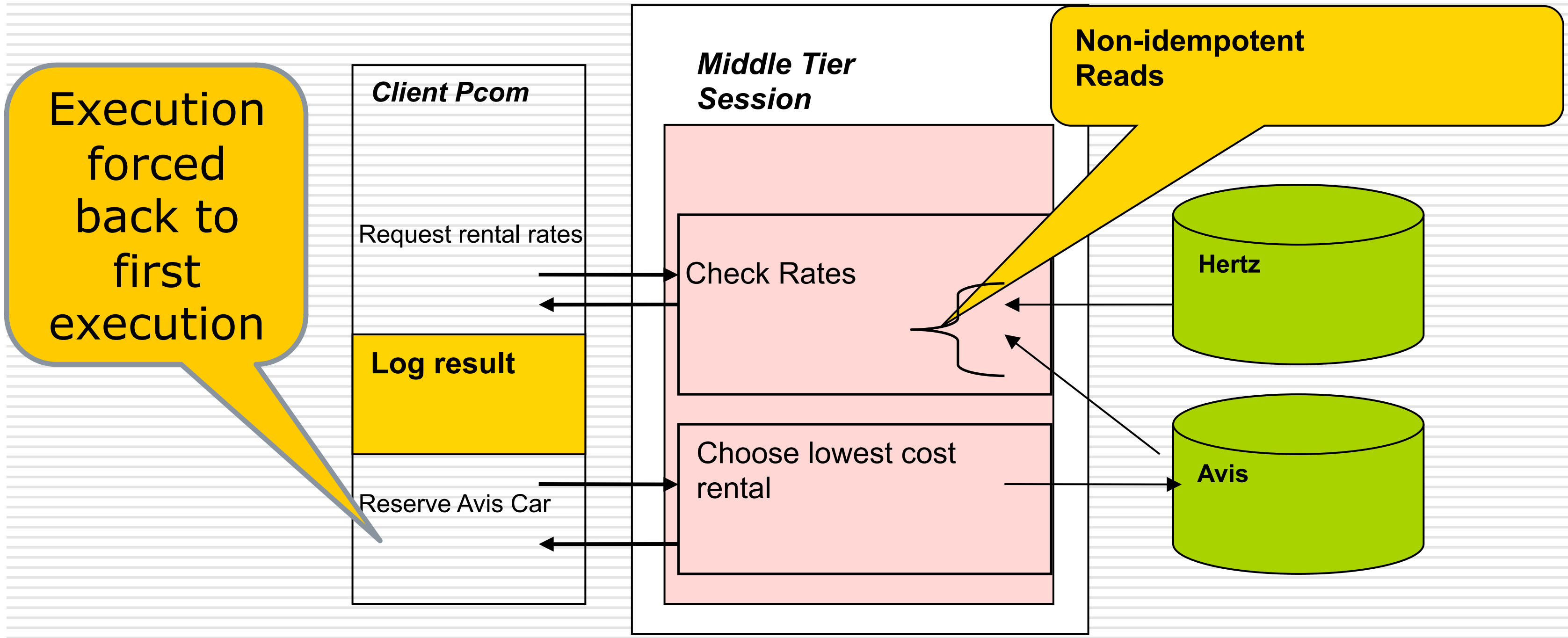
Wrap-up Reads

- **Let reads be last activity in LLcom**
  - Can have prior IR's and GIR's
  - No LLcom state changed by reads
- **Client logs result returned by LLcom**
- **Client execution on replay**
  - Determined by logged first reply
  - Not on replayed LLcom reply
    - Which may be different on every replay

LLcoms are NOT Idempotent But…



Wrap-up Read Example



A request is made to a car rental company, based on choosing the cheapest rates among those we have read. Logging for this choice is done at the client.

Summary

- Dependability requires everything
  - Availability
  - Scalability
  - Simple programming model
  - Excellent performance
  - Easy system administration
- Transparent stateful persistence
  - Highly desirable programming model
    - Natural for programmer- aiding correctness
    - Handling transaction errors within program
  - WITH Everything
    - Availability
    - Scalability
    - Simple programming model
    - Performance
    - Easy Admin

Bibliography: Phoenix

web page:
http://www.research.microsoft.com/research/db/phoenix/

- Lomet, D. Persistent Middle Tier Components without Logging. *IDEAS* Montreal, CA (July 2005)
- Lomet, D. Robust Web Services via Interaction Contracts. *TES'04 Workshop* Toronto, CA (Sept. 2004)
- Barga, R.,Lomet, D., Shegalov, G., and Weikum, G. Recovery Guarantees for Internet Applications. *ACM Trans. on Internet Technology* (August 2004)
- Barga, R., Chen, S. and Lomet, D. Improving Logging and Recovery Performance in Phoenix/App. *ICDE Conference,* Boston, MA (March 2004)
- Barga, R., Lomet, D., Paparizos, S., Yu, H., and Chandresekaran, S. Persistent Applications Via Automatic Recovery. *IDEAS Conference*, Hong Kong (July 2003)
- Barga, R., Lomet, D. Phoenix Project: Fault Tolerant Applications. *SIGMOD Record* 31, 2 (June 2002)
- Barga, R., Lomet, D. and Weikum, G. Recovery Guarantees for Multi-tier Applications. *ICDE Conference,* San Jose, CA (March 2002)
- Barga, R. and Lomet, D. Measuring and Optimizing a System for Persistent Database Sessions. *ICDE Conference,* Heidelberg, Germany (April 2001)
- Barga, R., Lomet, D., Baby, T., and Agrawal, S. Persistent Client-Server Database Sessions. *EDBT Conference,* Lake Constance, Germany (Mar. 2000)
- Barga,R. and Lomet, D. Phoenix: Making Applications Robust.(demo paper) *ACM SIGMOD Conference*, Philadelphia, PA (June, 1999)