Distributed Self-Propelled Instrumentation

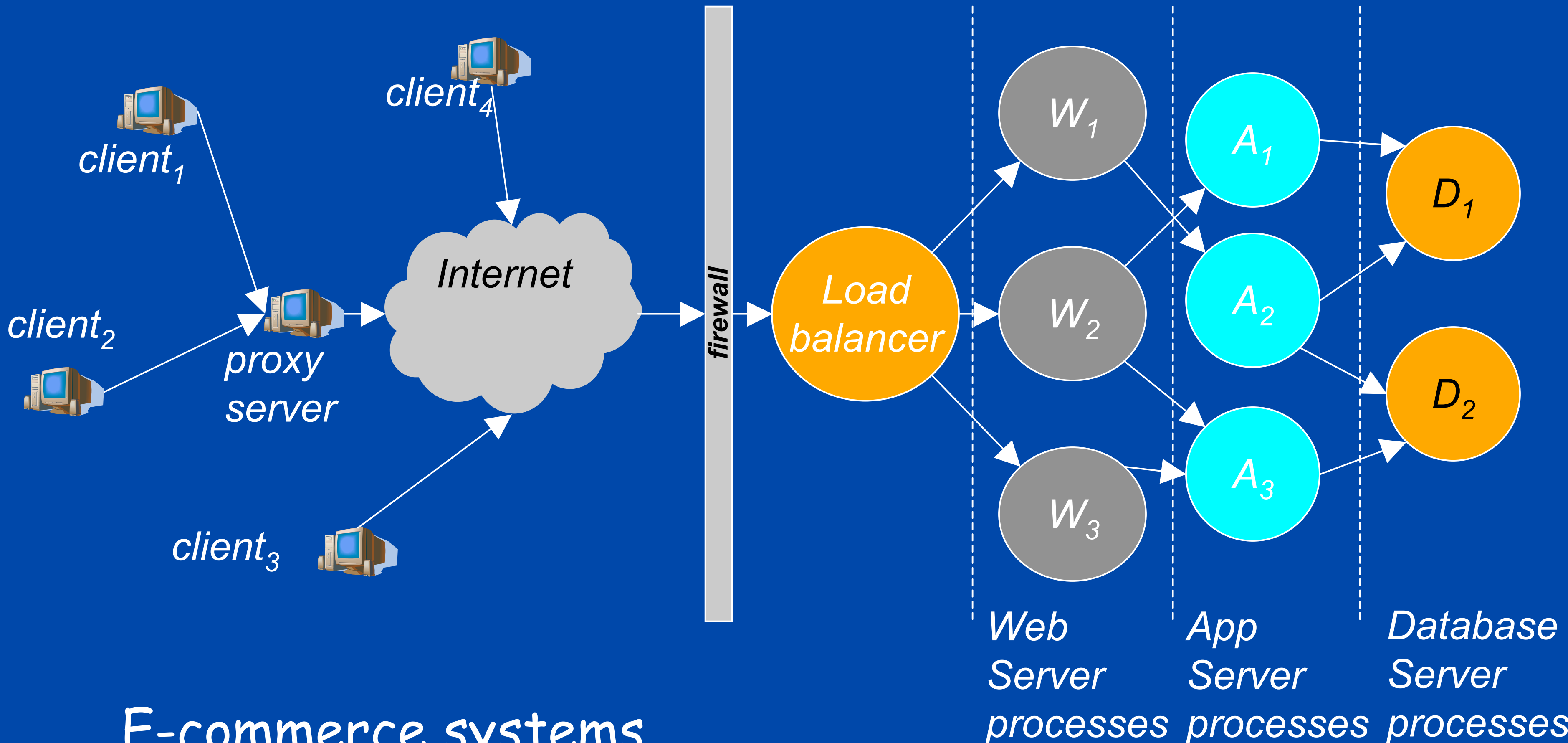
Alex Mirgorodskiy
VMware, Inc.
Barton P. Miller
University of Wisconsin-Madison

Motivation

Diagnosis of production systems is hard

- Problems are difficult to reproduce
  - Intermittent or environment-specific (anomalies)
  - "Rare but dangerous"
- Systems are large collections of black boxes
  - Many distributed components, different vendors
  - Little support for monitoring/debugging
- Collected data are difficult to analyze
  - High volume
  - High concurrency

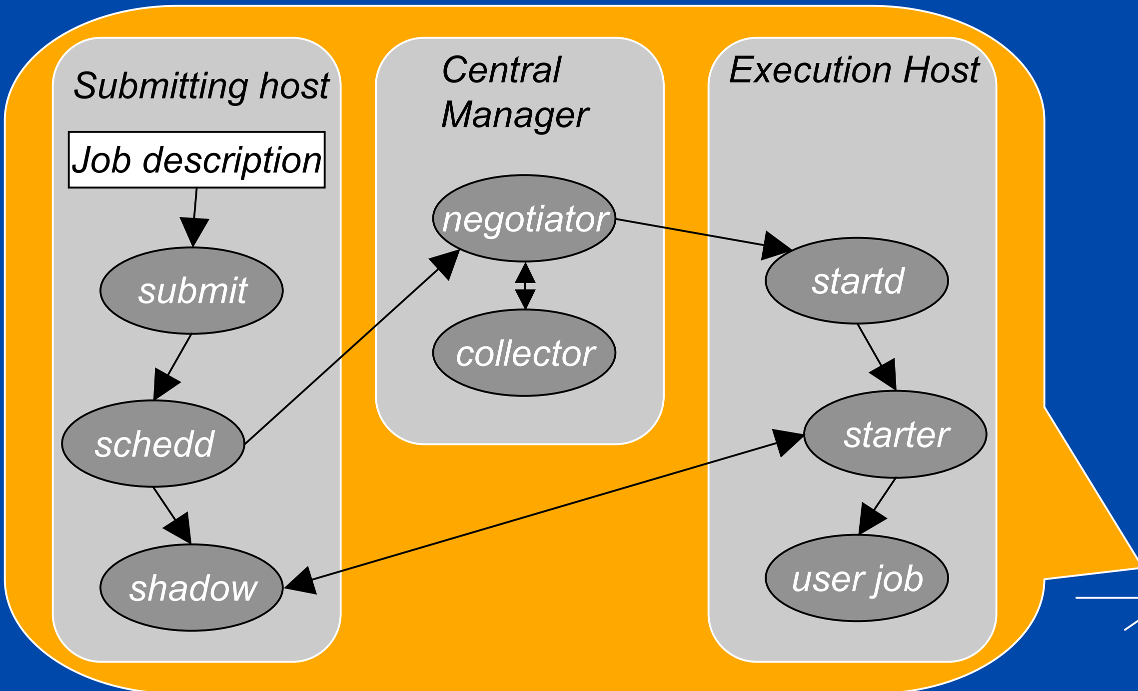
Common Environments

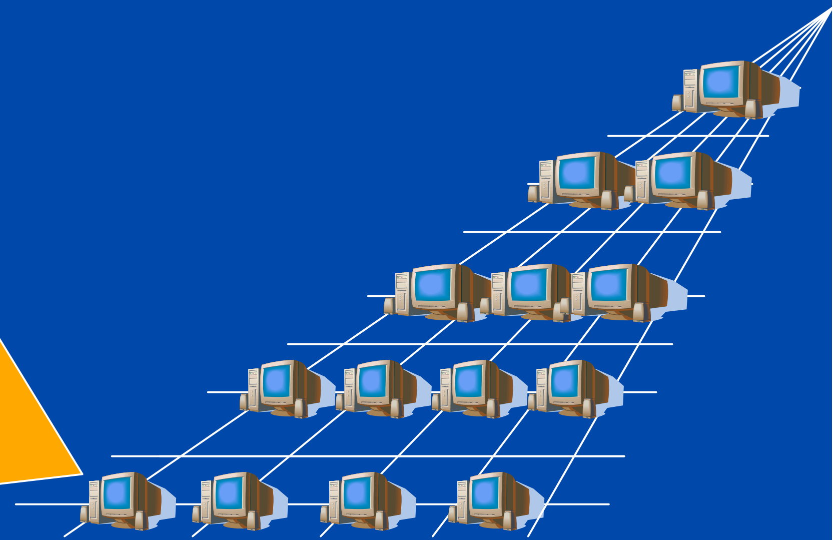


E-commerce systems

- Multi-tier: Clients, Web, DB servers, Business Logic
- Hard to debug: vendors have SWAT teams to fix bugs
  - Some companies get paid $1000/hour

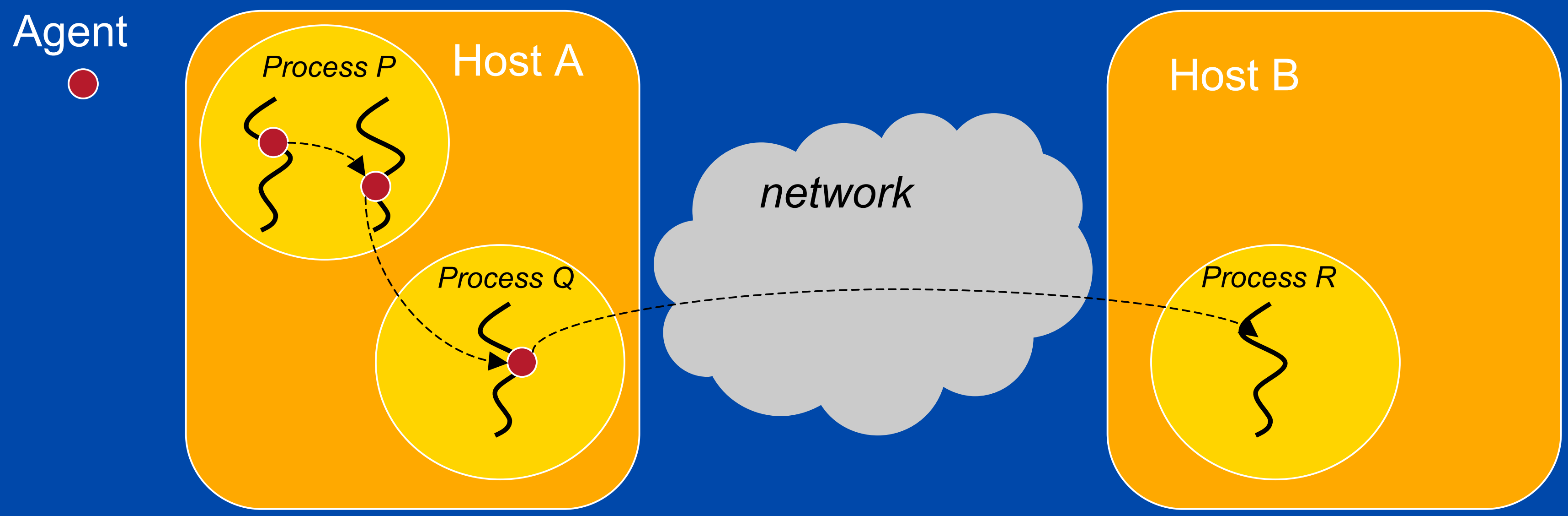
Common Environments





- Clusters and HPC systems
  - Large-scale: failures happen often (MTTF: 30 – 150 hours)
  - Complex: processing a Condor job involves 10+ processes
- The Grid: Beyond a single supercomputer
  - Decentralized
  - Heterogeneous: different schedulers, architectures
- Hard to detect failures, let alone debug them

Approach

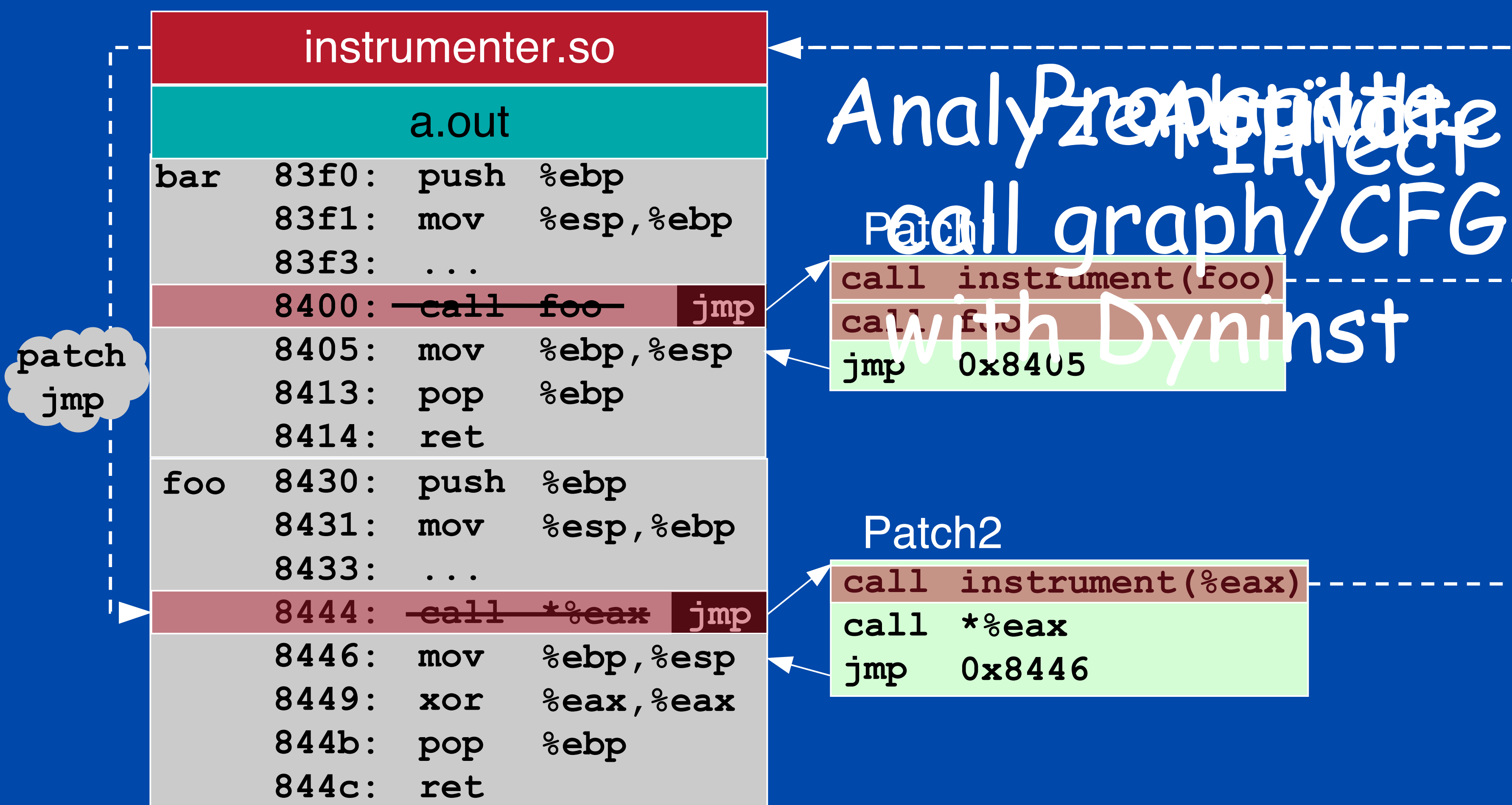


- User provides activation and deactivation events
- Agent propagates through the system
  - Collects distributed control-flow traces
- Framework analyzes traces automatically
  - Separates traces into flows (e.g., HTTP requests)
  - Identifies anomalous flows and the causes of anomalies

Self-Propelled Instrumentation: Overview

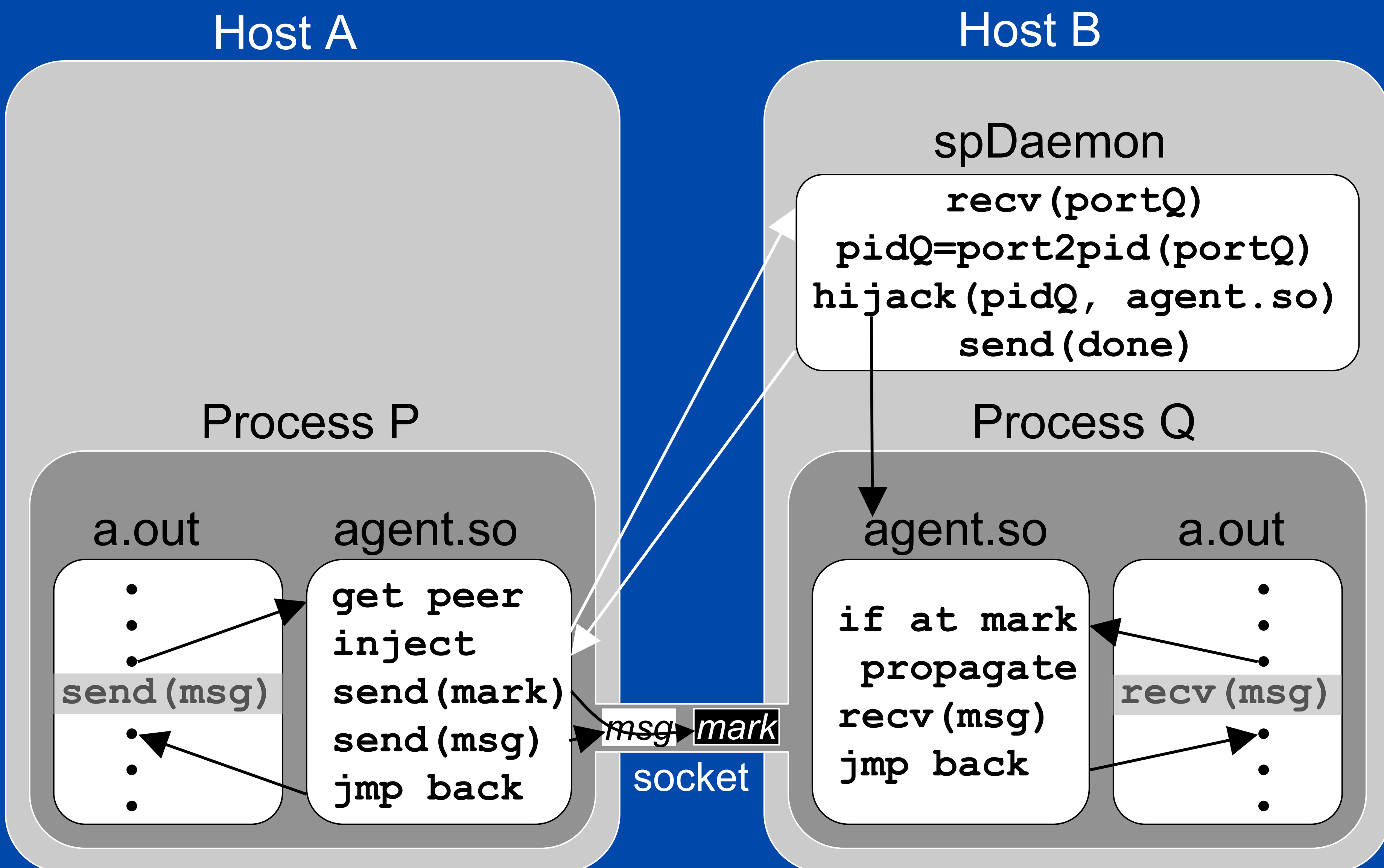
- The agent sits inside the process
  - Agent = small code fragment
- The agent propagates through the code
  - Receives control
  - Inserts calls to itself ahead of the control flow
  - Crosses process, host, and kernel boundaries
  - Returns control to the application
- Key features
  - On-demand distributed deployment
  - Application-agnostic distributed deployment

Within-process Propagation



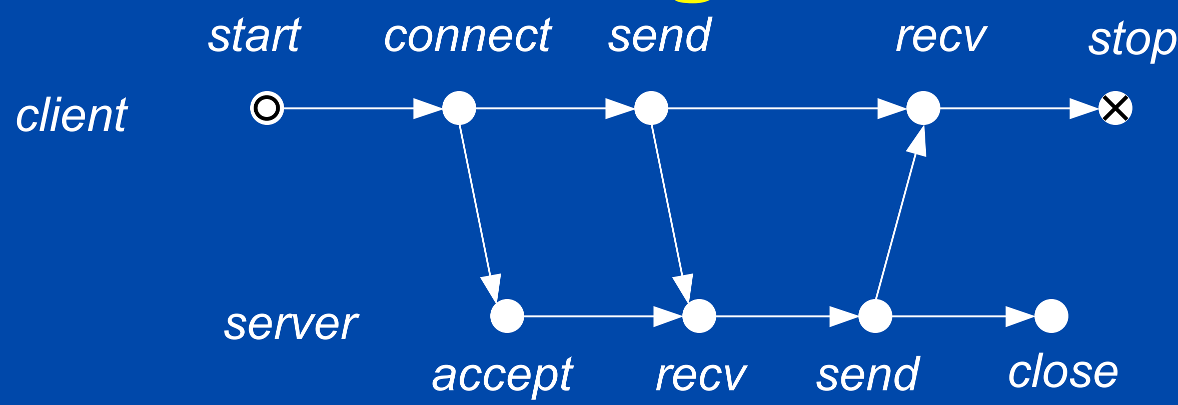
Dynamic, low-overhead control flow tracing

Cross-process Propagation



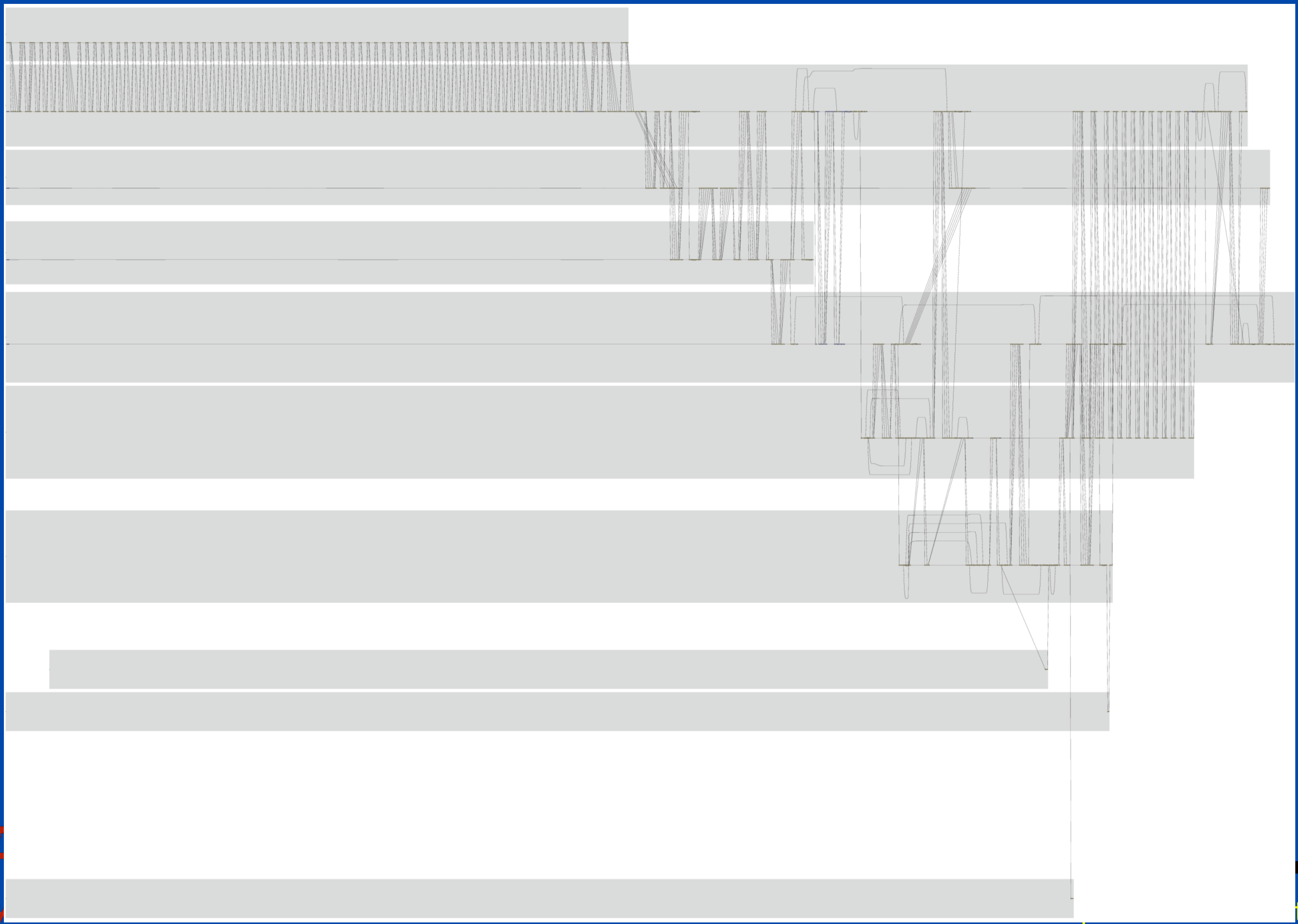
On-demand distributed deployment

PDG for a Simple Socket Program



- PDG: Parallel Dynamic Program Dependence Graph
  - Nodes: observed events
  - Intra-process edges: link consecutive events
  - Cross-process edges: link sends with matching recvs
- PDGs from real systems are more complex

PDG for One Condor Job



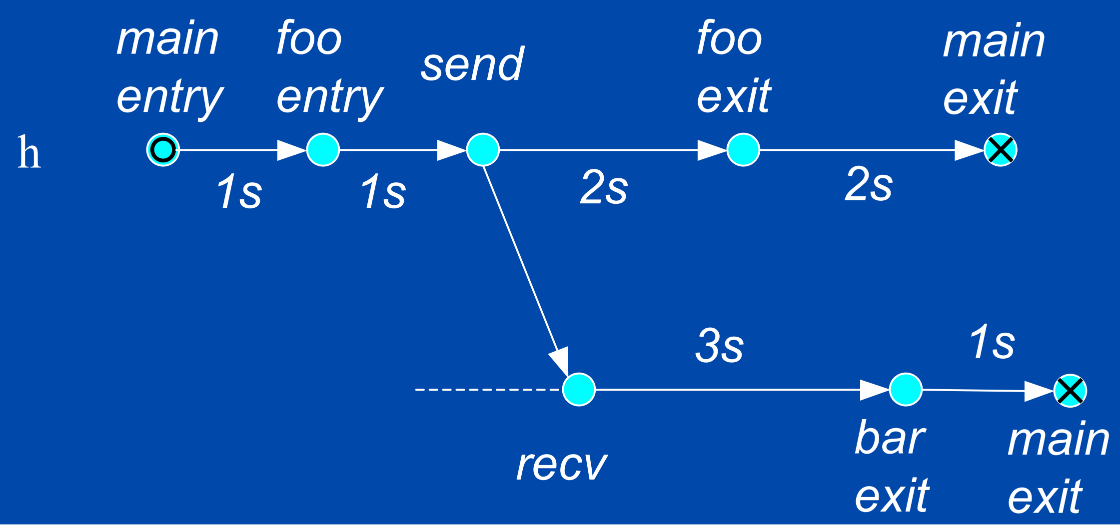
Automated Diagnosis

- Challenge for manual examination
  - High volume of trace data
- Automated Approach: find anomalies
  - Normal behavior often is repetitive
  - Pathological problems often are easy to find
  - Focus on anomalies: infrequent bad behavior

Overview of the Approach

- Obtain a collection of control flows
  - E.g., per-request traces in a Web server
- Anomaly detection: find an unusual flow
  - Summarize each flow as a profile
  - Assign suspect scores to profiles
- Root cause analysis: find why a profile is anomalous
  - Function responsible for the anomaly

Anomaly Detection: Distributed Profiles

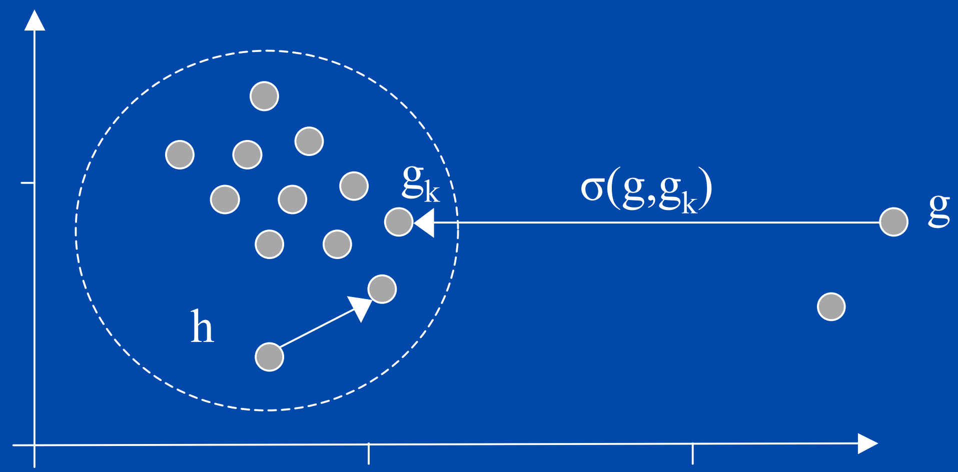


t_{main} t_{foo} t_{bar}

$$p^t(h) = <0.4, 0.3, 0.3>$$

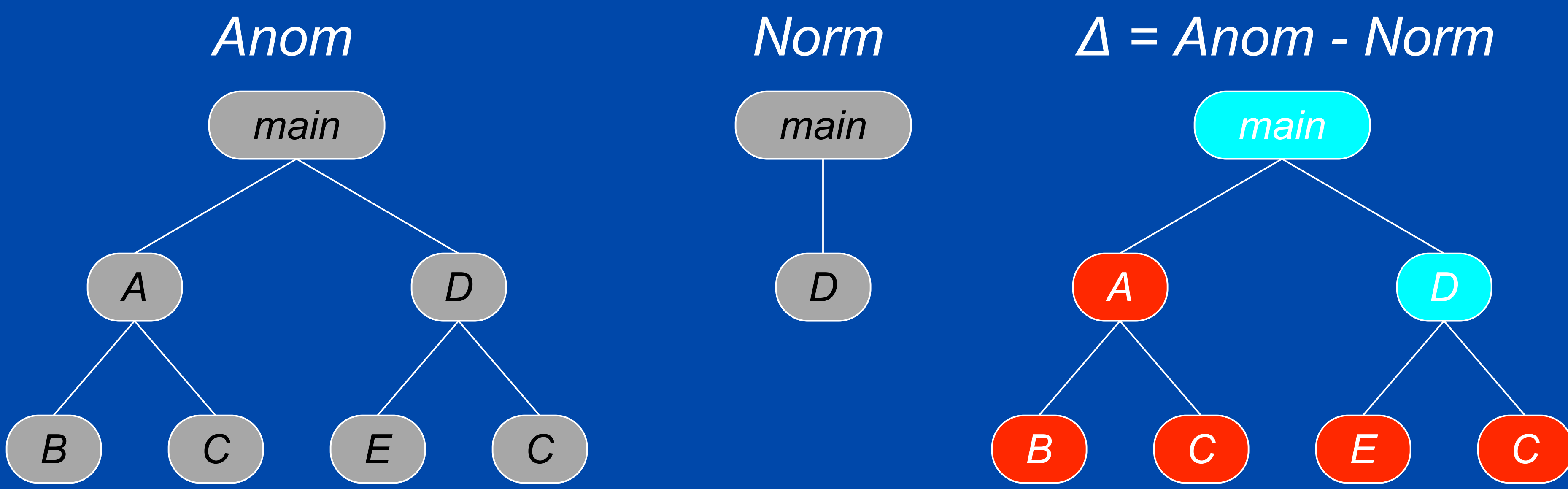
Time	$p^t = <t_1, \ldots, t_F>$	t_i = normalized time spent in function f_i
Communication	$p^s = <s_1, \ldots, s_F>$	s_i = normalized number of bytes sent by f_i
Composite	$p^c = <t_1,\ldots, s_1,\ldots>$	Concatenate p^t and p^s
Coverage	$p^v = <v_1, \ldots, v_F>$	v_i = 1 if function f_i was called; v_i = 0 otherwise

Anomaly Detection: Suspect Scores



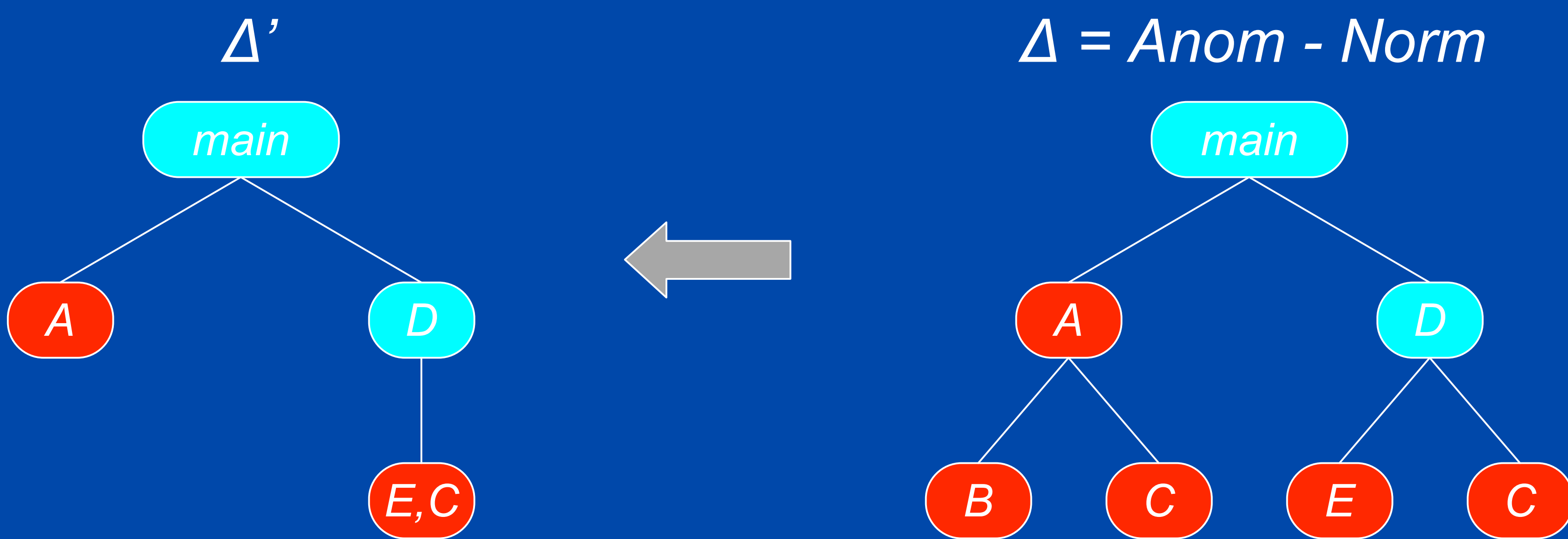
- $\sigma(g)$ = distance to a common or known-normal node
- Can detect multiple anomalies
- Does not require known examples of prior runs
  - Unsupervised algorithm
- Can use such examples for fewer false positives
  - One-class ranking algorithm

Finding the Cause: Coverage Analysis



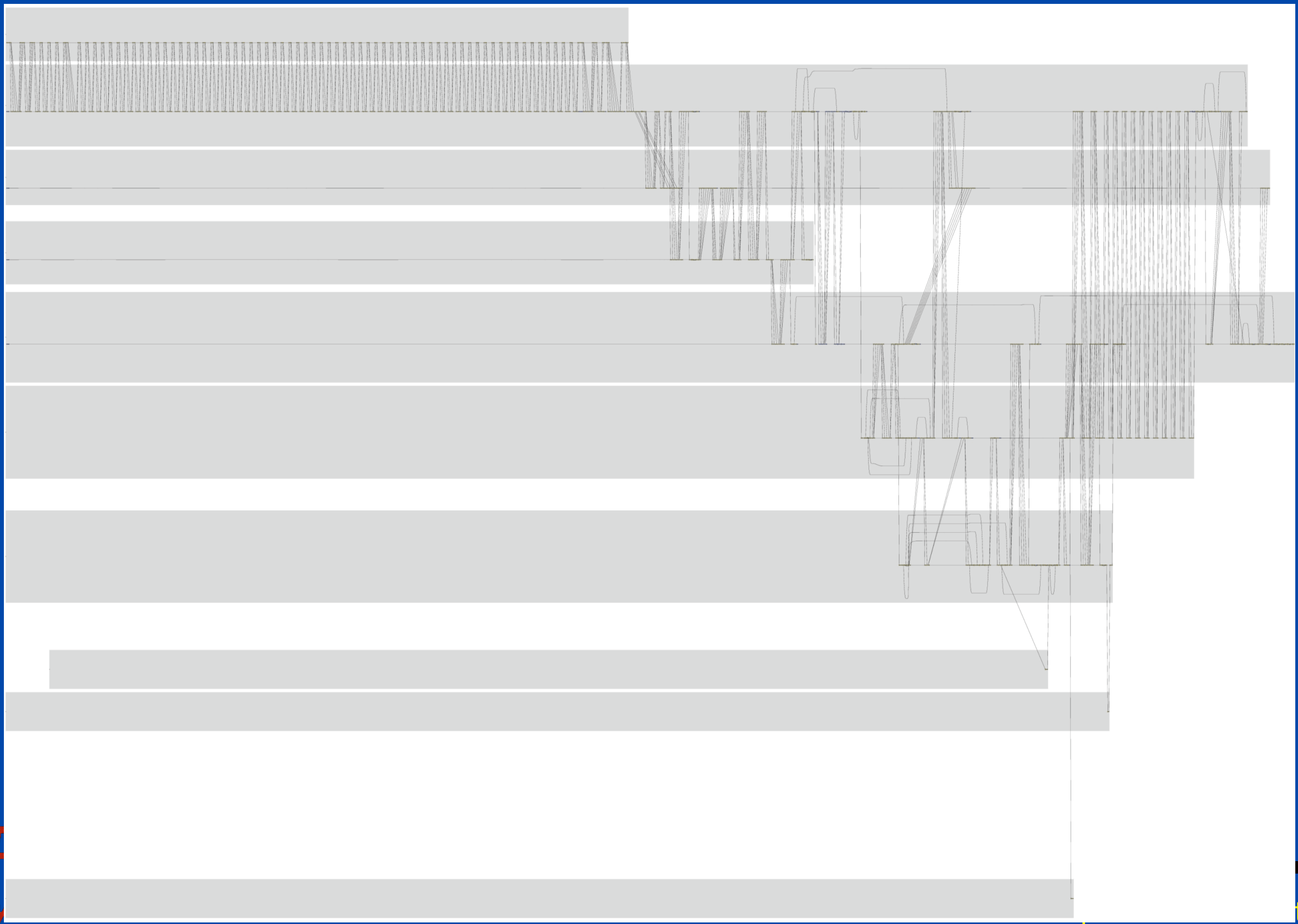
- Find call paths taken only in the anomalous flow
  - Δ = {main→A, main→A→B, main→A→C, main→D→E, main→D→C}
- Correlated with the failure
- Likely location of the problem

Finding the Cause: Coverage Analysis

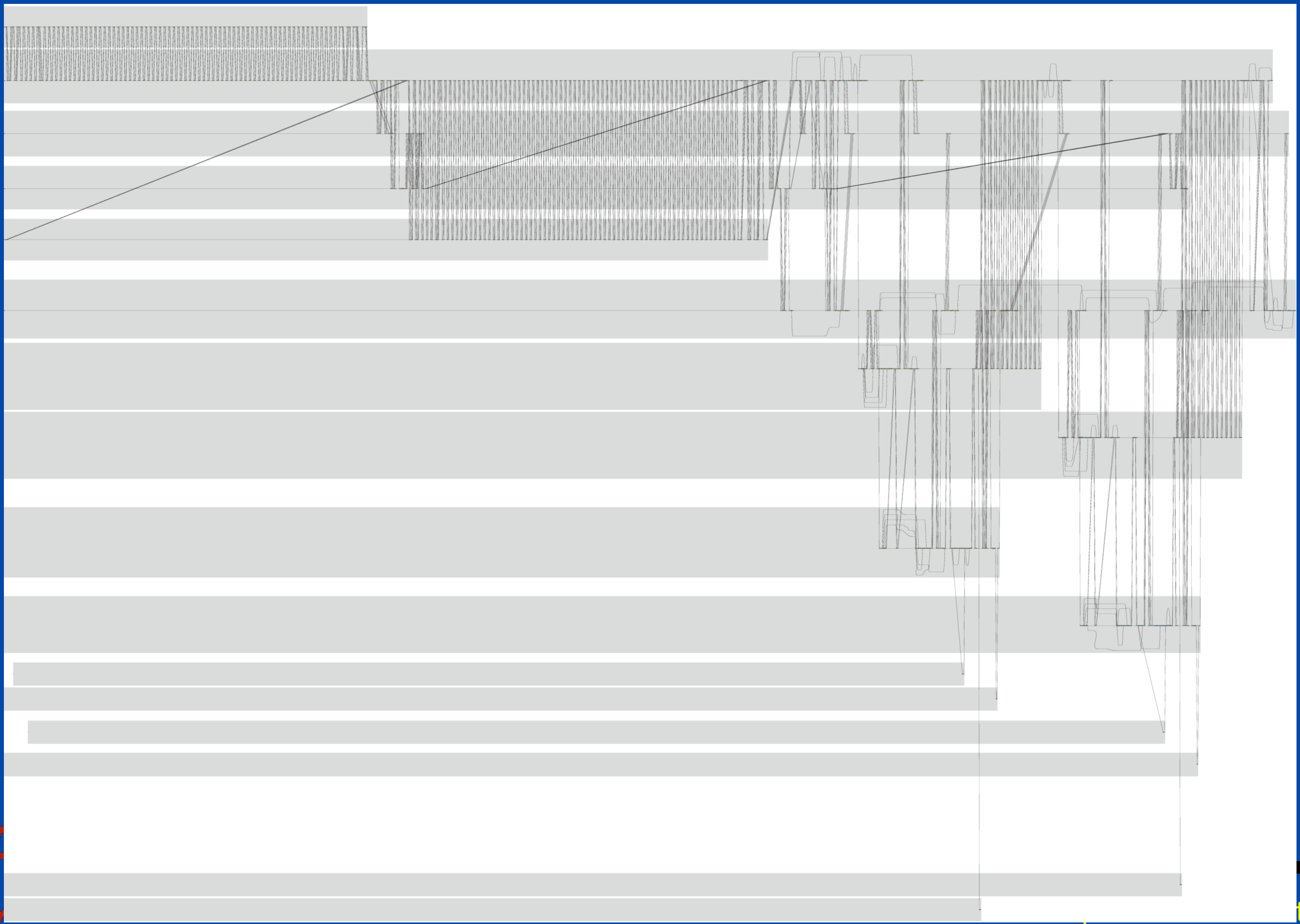


- Limitation of coverage analysis: too many reports
  - Noise in the trace, different input, workload
- Can eliminate effects of earlier differences
  - Retain the shortest prefixes in Δ
  - Merge leaves
- Can rank paths by the time of occurrence or length
  - Put the cause ahead of the symptoms or simplify manual examination

PDG for One Condor Job



PDG for Two Condor Jobs

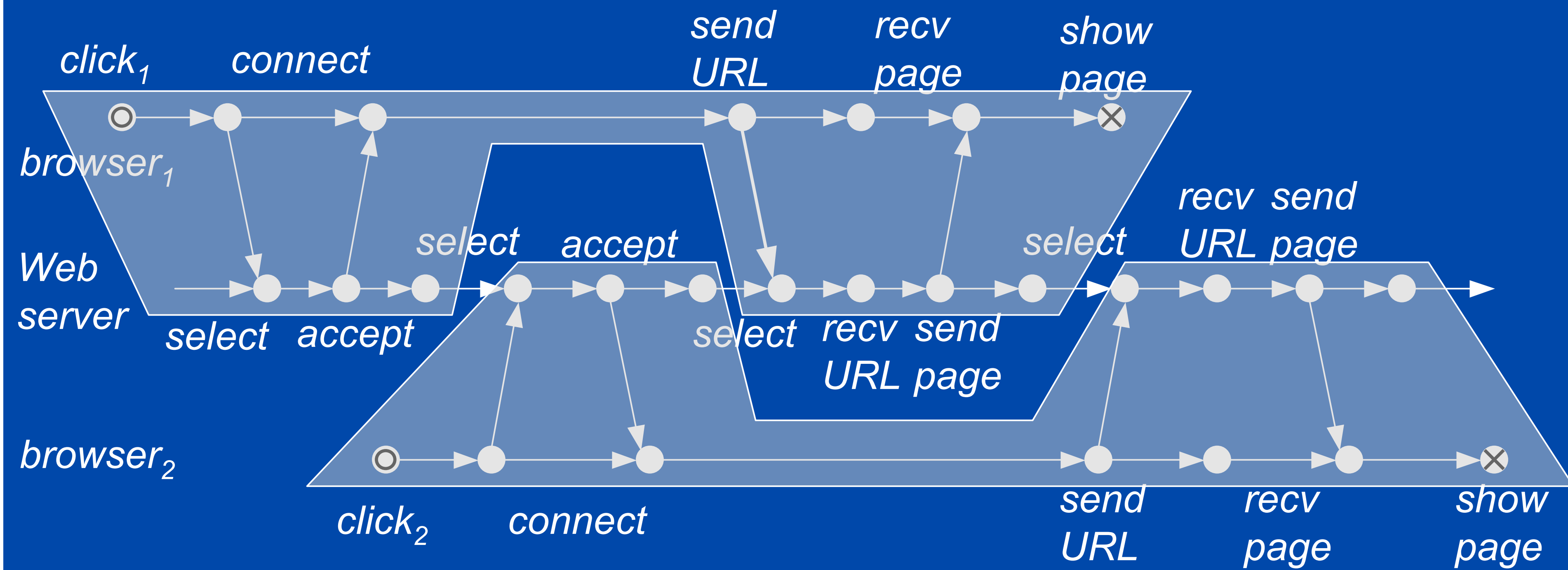


tation

Separating Concurrent Flows

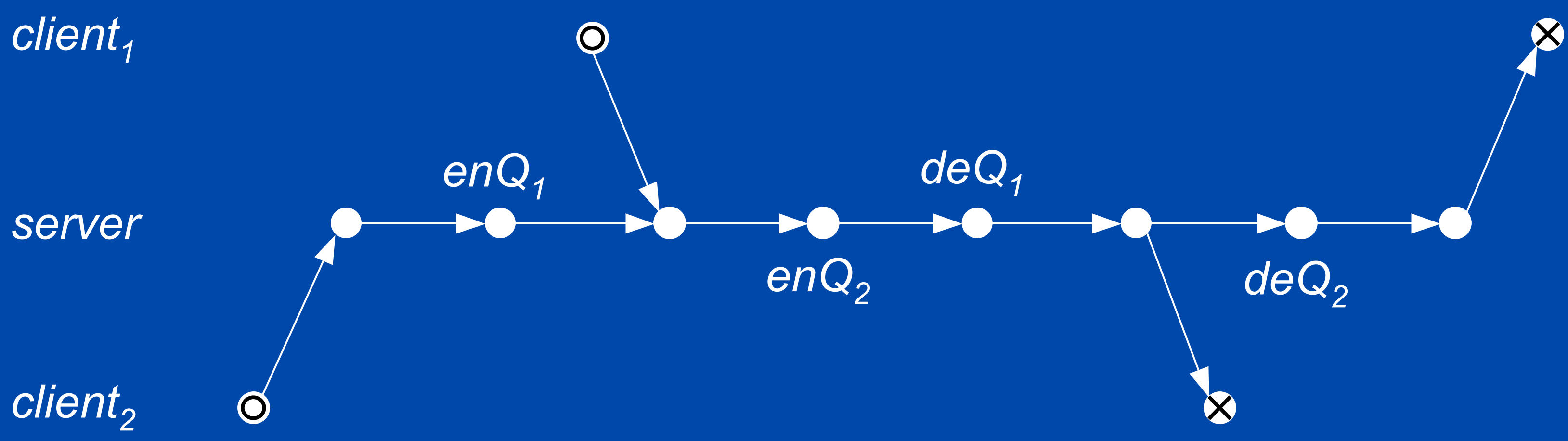
- Concurrency produces interleaved traces
  - Servers switch from one request to another
- Analyzing interleaved traces is difficult
  - Irrelevant details from other users
  - High trace variability → everything is an anomaly
- Solution: separate traces into flows

Flow-Separation Algorithm



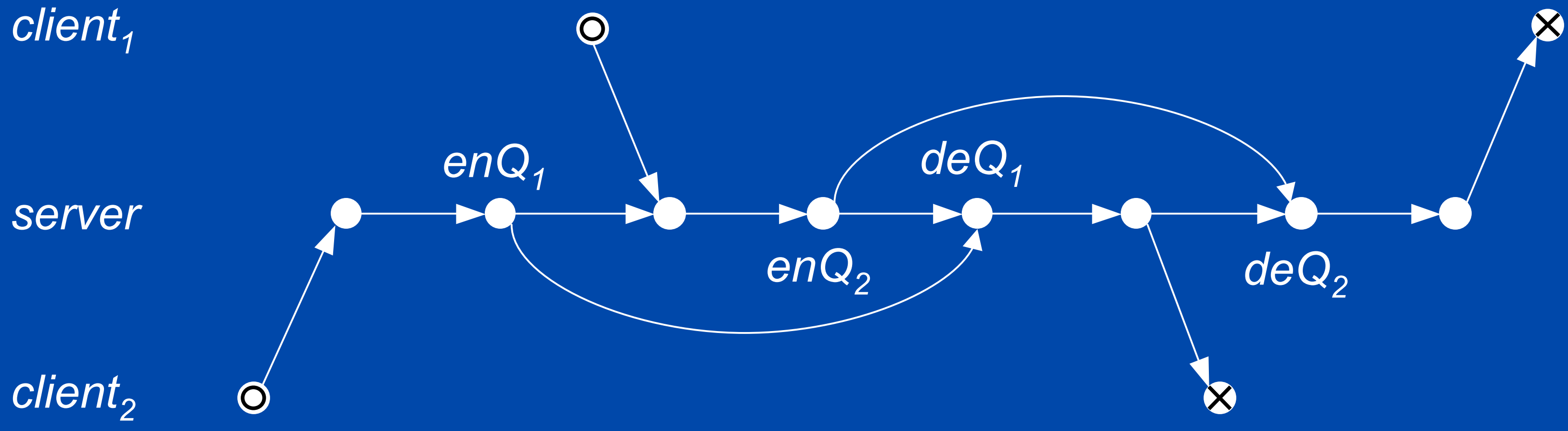
- Decide when two events are in the same flow
  - (send → recv) and (local → non-recv)
- Remove all other edges
- Flow = events reachable from a start event

Limitation



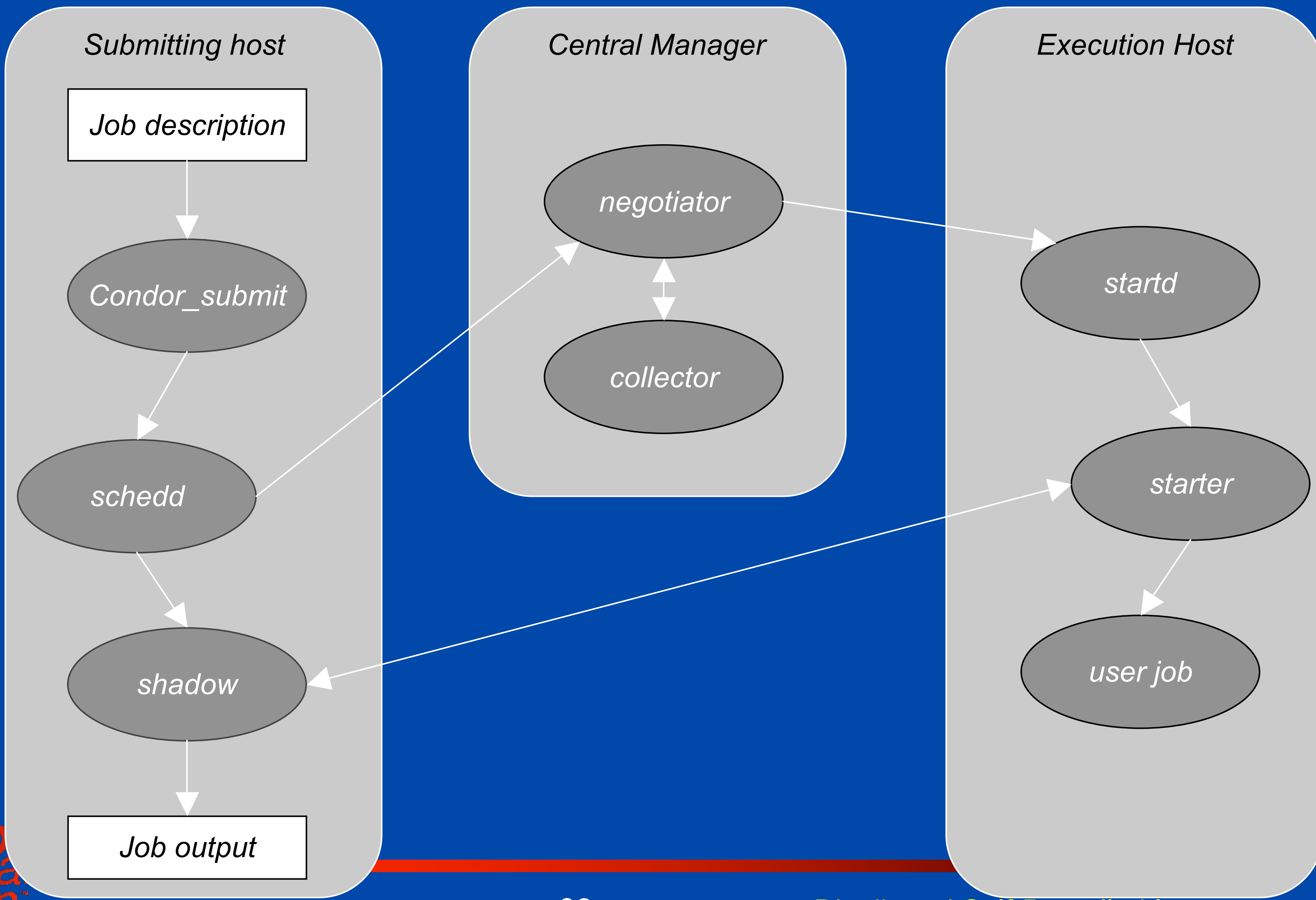
- Rules violated for programs with queues
  - enQ_1 and deQ_1 must belong to the same flow
  - Assigned to different flows by our application-independent algorithm

Addressing the Limitation: Directives



- Pair events using <evt,joinattr> custom directives
- Evt: location in the code
- Joinattr: related events have equal attr values

Experimental Study: Condor



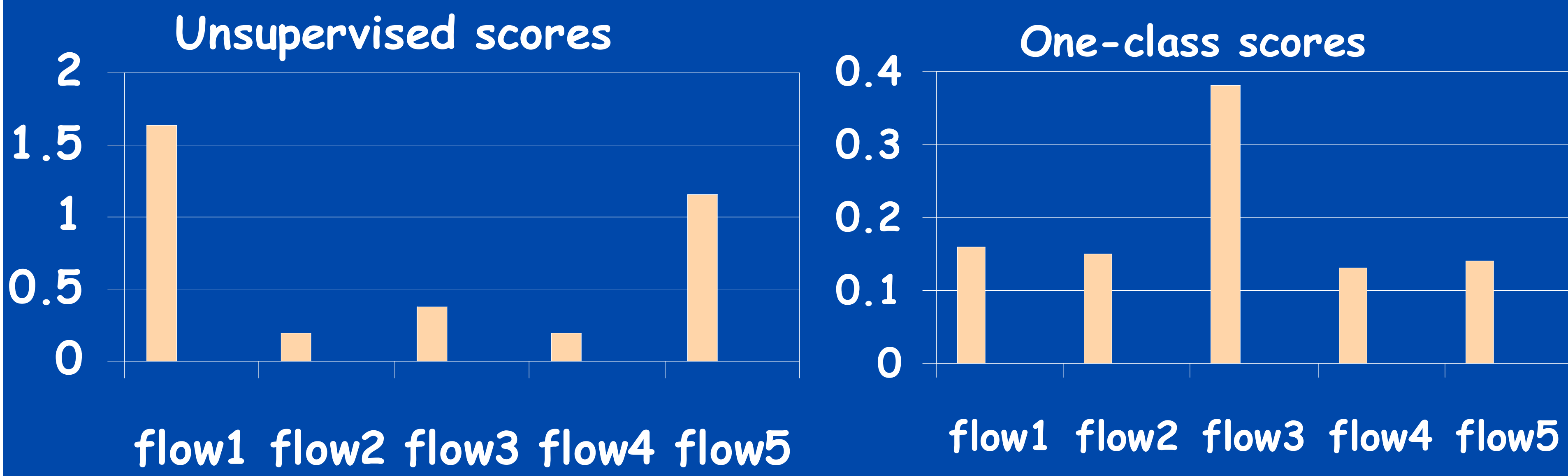
Job-run-twice Problem

- Fault handling in Condor
  - Any component can fail
  - Detect the failure
  - Restart the component
- Bug in the shadow daemon
  - Symptoms: user job ran twice
  - Cause: intermittent crash after shadow reported successful job completion

Debugging Approach

- Insert an intermittent fault into shadow
- Submit a cluster of several jobs
  - Start tracing condor_submit
  - Propagate into schedd, shadow, collector, negotiator, startd, starter, mail, the user job
- Separate the trace into flows
  - Processing each job is a separate flow
- Identify anomalous flow
  - Use unsupervised and one-class algorithms
- Find the cause of the anomaly

Finding Anomalous Flow



- Suspect scores for composite profiles
- Without prior knowledge, Flows 1 and 5 are unusual
  - Infrequent but normal activities
  - Use prior known-normal traces to filter them out
- Flow 3 is a true anomaly

Finding the Cause

- Computed coverage difference
  - 900+ call paths
- Filtered the differences
  - 37 call paths left
- Ranked the differences
  - 14th path by time / 1st by length as called by schedd:

    main
    → DaemonCore::Driver
    → DaemonCore::HandleDC_SERVICEWAITPIDS
    → DaemonCore::HandleProcessExit
    → Scheduler::child_exit
    → DaemonCore::GetExceptionString
  - Called when shadow terminates with a signal
- Last function called by shadow = failure location

Conclusion

- Self-propelled instrumentation
  - On-demand, low-overhead control-flow tracing
  - Across process and host boundaries
- Automated root cause analysis
  - Finds anomalous control flows
  - Finds the causes of anomalies
- Separation of concurrent flows
  - Little application-specific knowledge

Related Publications

- A.V. Mirgorodskiy and B.P. Miller, “Diagnosing Distributed Systems with Self-Propelled Instrumentation", Under submission,
  - ftp://ftp.cs.wisc.edu/paradyn/papers/Mirgorodskiy07DistDiagnosis.pdf
- A.V. Mirgorodskiy, N. Maruyama, and B.P. Miller, “Problem Diagnosis in Large-Scale Computing Environments", SC'06, Tampa, FL, November 2006,
  - ftp://ftp.cs.wisc.edu/paradyn/papers/Mirgorodskiy06ProblemDiagnosis.pdf
- A.V. Mirgorodskiy and B.P. Miller, "Autonomous Analysis of Interactive Systems with Self-Propelled Instrumentation", *12th Multimedia Computing and Networking (MMCN 2005),* San Jose, CA, January 2005,
  - ftp://ftp.cs.wisc.edu/paradyn/papers/Mirgorodskiy04SelfProp.pdf