



Cornell University

Raytheon

Using Self-Regenerative Tools to Tackle Challenges of Scale



Johannes Gehrke

Cornell SRS Team



The QuickSilver Team

- At Cornell:
 - Birman/van Renesse: Core platform
 - Gehrke: Content filtering technology
 - Francis: Streaming content delivery
- At Raytheon:
 - Lala/Bracewell: Military scenarios (DD(X), SPY-3)
 - Technology transfer. Target: FORCEnet, JBI, NCW components
 - Metrics
- With help from:
 - AFRL JBI team in Rome NY



Talk Outline

- **Technical approach**
- Components
 - Scalable reliable multicast
 - Scalable event processing
 - Large-scale, real-time delivery
- Summary and outlook



Technical Approach

- **Objective: Overcome communications challenges that plague (and limit) current GIG/NCES platforms**
 - Dramatically improve time-critical event delivery delays, speed of event filtering layer
 - Do this even when sustaining damage or when under attack
- **Existing COTS publish-subscribe technology, particularly in Web Services (SOA) platforms:**
 - Not designed for these challenging settings. Scale poorly.
 - Forces military to “hack around” limitations, else major projects can stumble badly
- **Problem identified by Air Force JBI team in Rome NY**
 - But also a major concern for companies like Amazon



Technical Approach (Contd.)

- **GIG/NCES vision centers on reliable communication protocols, like publish-subscribe.**
 - Underlying protocols are old... hit limits 15 years ago!
 - Faster hardware has helped... but only a little
- **Peer-to-peer epidemic protocols ("gossip") have never been applied in such systems**
 - Makes our system robust, self-repairing, and scalable
- **Existing systems take an all-or-nothing approach to reliability. Under stress, we often get nothing.**
 - Probabilistic guarantees enables better solutions
 - But need provable guarantees of quality



Time-Critical Eventing

- Eventing: Publishers publish events, which are then received by **subscribers** that have posed **queries**
- Requirements:
 - Scalability
 - Graceful degradation under failures
 - Resilience under attack
 - Probabilistic real-time guarantees



Components

- Scalable reliable multicast
- Scalable event processing
- Large-scale, real-time delivery



Status: January 2005

- Scalable reliable multicast
 - Baseline numbers
- Large-scale, real-time delivery
 - Baseline numbers
- Scalable event processing
 - Baseline numbers



Status: July 2005

- Scalable reliable multicast: Slingshot
 - One group: > 2-3 orders of magnitude latency reduction (real implementation)
 - Many groups: > 20 times latency reduction (simulation)
- Scalable event processing: Cayuga
 - Event processing: 2-3 orders of magnitude higher throughput (real implementation)
- Large-scale, real-time delivery: ChunkySpread
 - Factor of 5 increase in throughput (simulation)



Talk Outline

- Technical approach
- Components
 - Scalable reliable multicast
 - Scalable event processing
 - Large-scale, real-time delivery
- Summary and outlook



Problem Definition

- One sender, many receivers
- Key performance metrics
 - Throughput: sustainable event rate
 - Reliability: what % of messages get through?
 - Delay: time (ms) from event injection to delivery
 - Impact of scale and failures: tolerate failures and run in big settings without degradation in metrics



The Baseline

- State of the art: Scalable Reliable Multicast (SRM)
 - Probably the best “scalable” technology today, although not available in products
 - Was developed in DARPA FTN effort
- SRM uses NAKs to discover packet loss; hence discovery time is proportional to time between multicasts at sender (t)

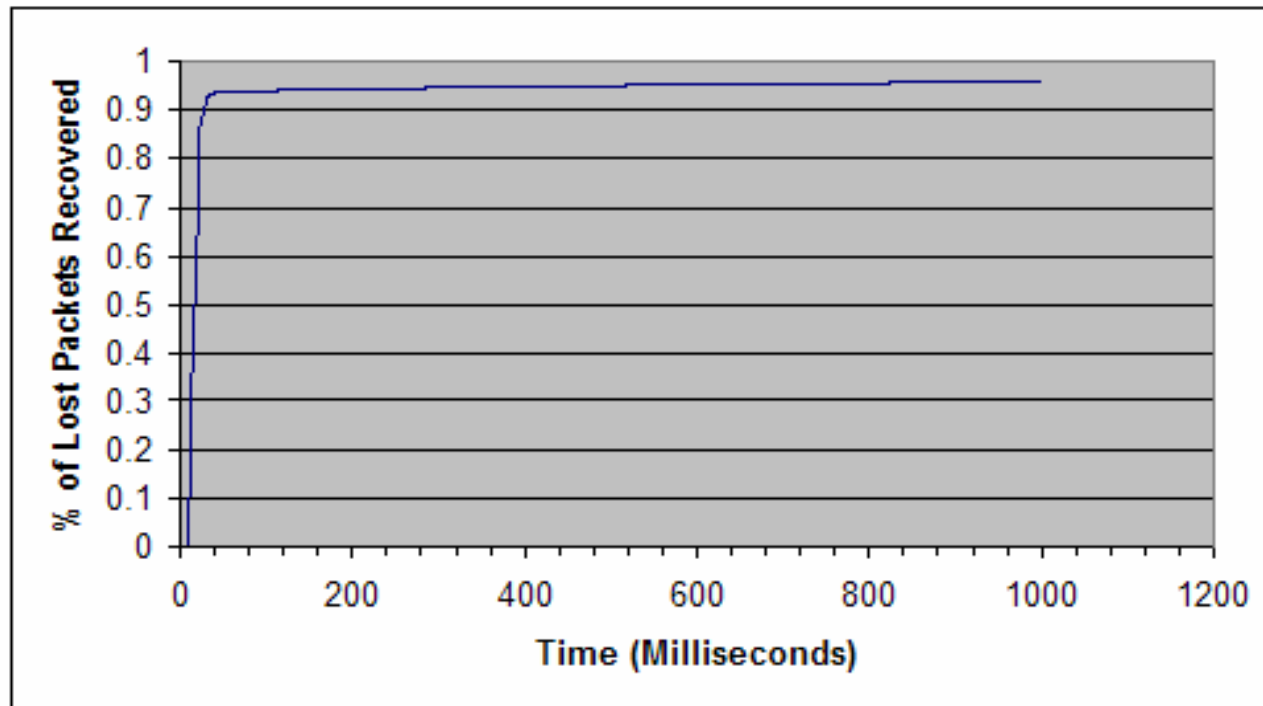


Experiment Setup

- 20 nodes on Emulab testbed (Utah)
- Loss Rate fixed at 1%: packets dropped at end buffers
- All nodes send and receive
- Inter-node Latencies: 100-200 microseconds
- Total multicast traffic of 1000 packets/sec: fragmented into different groups
- Why do groups affect performance?
Inter-send time t : For one group, a packet is sent by each node every 20 ms; for n groups, every $n * 20$ ms. t increases linearly with # of groups



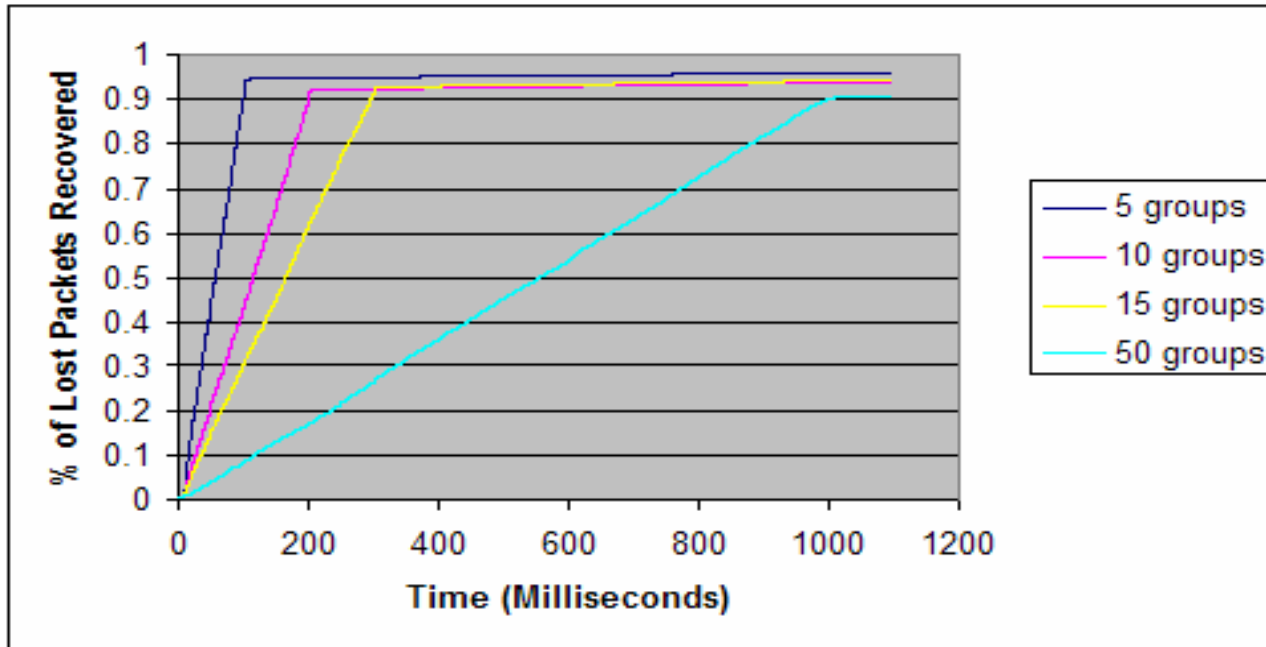
SRM Time-Criticality



- Most packets recovered before the 'neck', at 40 ms: $t + c$
 - t is inter-send time (20 ms for single group)
 - c is constant time taken by sender to retransmit (20 ms here)



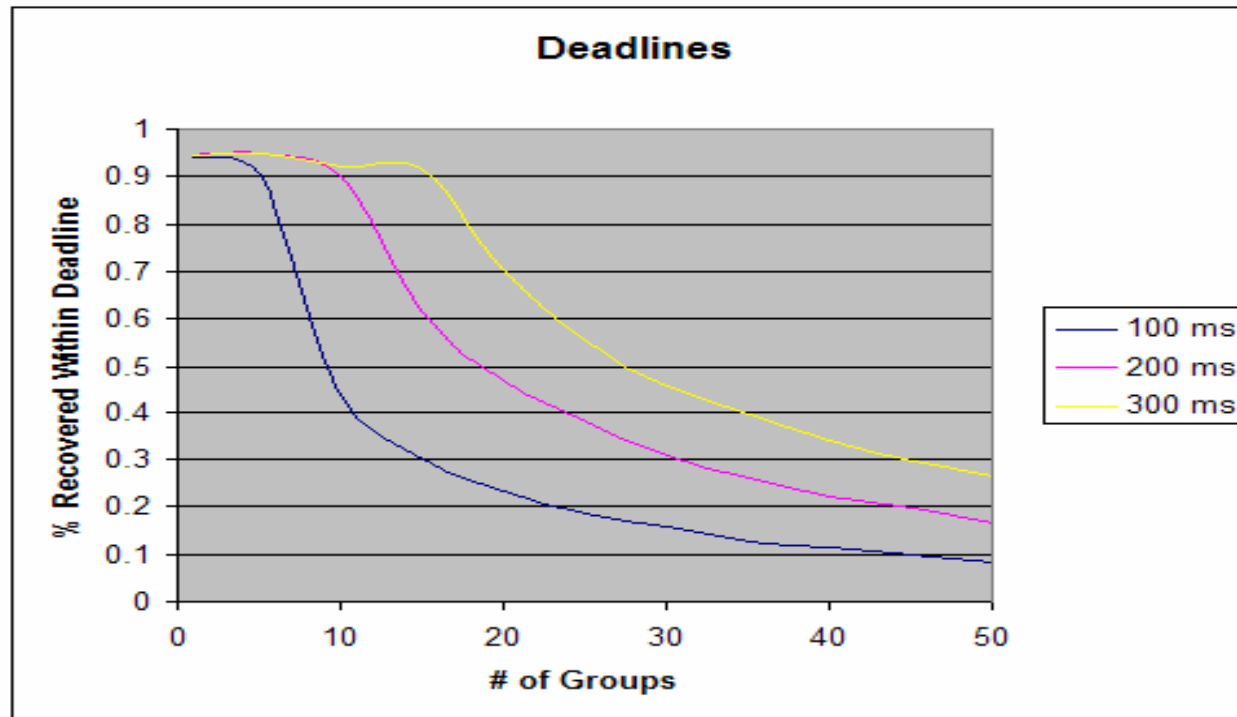
Multiple Groups



- Inter-send time t increases with # of groups; recovery 'neck' stretches out



Deadlines



- SRM works well only when deadlines are within $t + c$.
e.g: for 10 groups, $t + c = 220$ ms. At 10 groups, SRM performs well for 200 and 300 ms, and badly for 100 ms

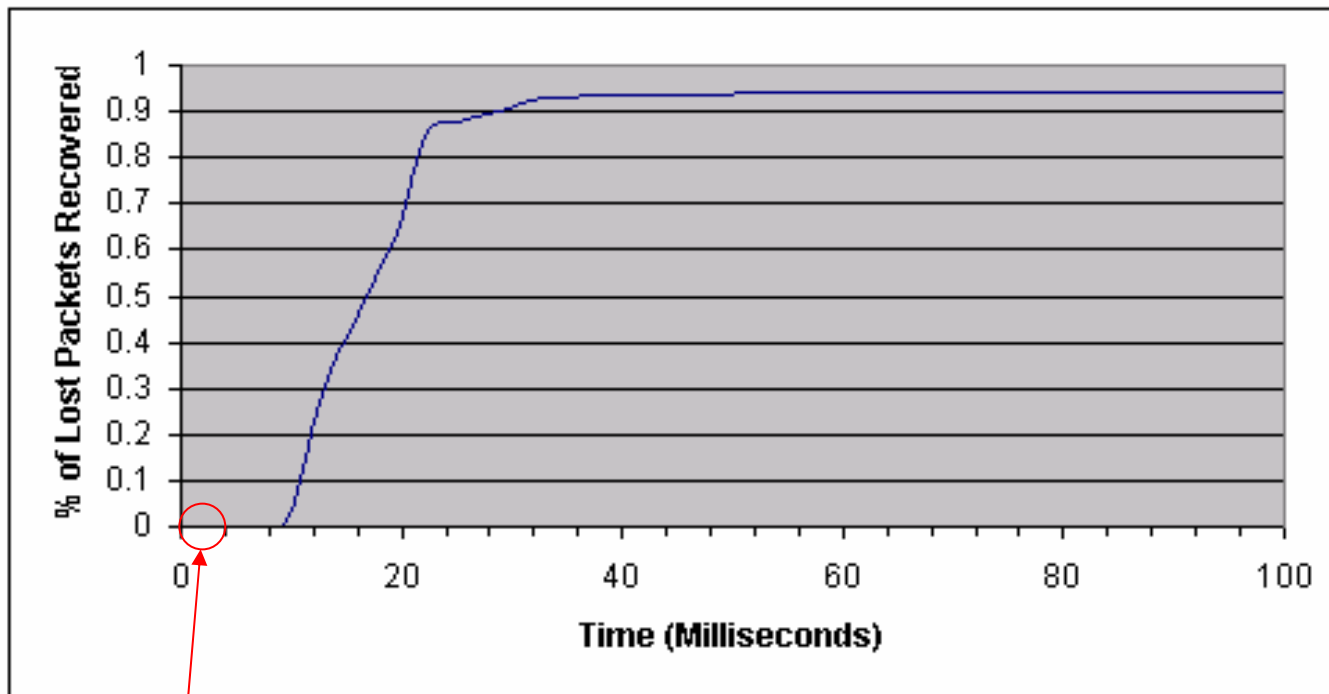


Note

- This is not only an issue of the number of groups!



SRM Time-Criticality: Close-Up



- Slingshot lives here: recovers ~90% within 3-4 RTTs, or 2 ms!



Baseline: Summary

- SRM time-critical properties do not scale well in numbers of groups
- Even in a single group its recovery/discovery time of packets is limited by the inter-send time
- Slingshot (Receiver-based FEC) should achieve recovery times that are 2-3 orders of magnitude better



Talk Outline

- Technical approach
- Components
 - Scalable reliable multicast
 - Baseline
 - Slingshot
 - Scalable event processing
 - Large-scale, real-time delivery
 - (Scalable replication)
- Summary and outlook



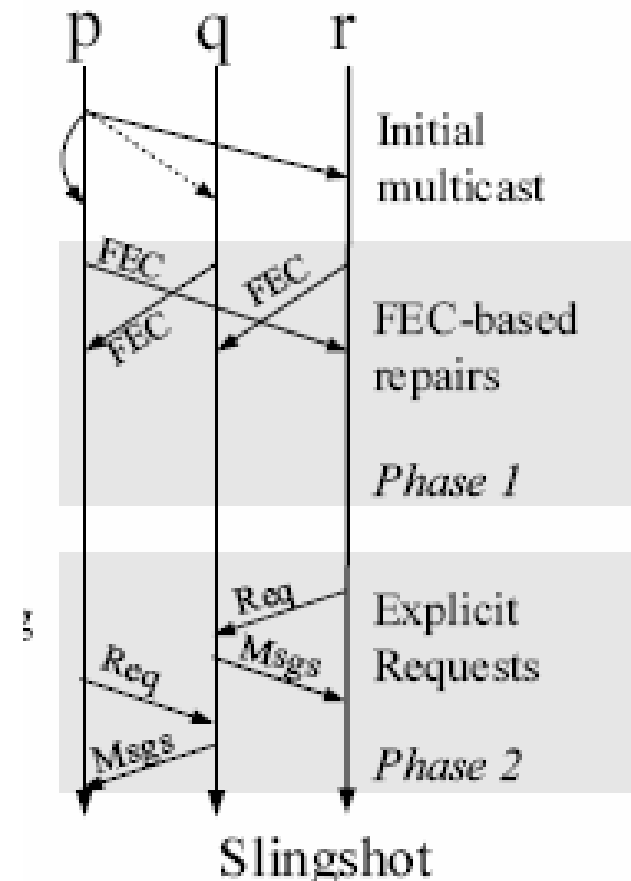
Slingshot

- Time-critical event notification protocol
 - Idea: *probabilistic* real-time goals
 - Pay a higher overhead but reduce frequency of missed deadlines
- Probabilistic Guarantees:
With $x\%$ overhead, $y\%$ data is delivered within t seconds.



Slingshot Protocol

- Three phases:
 - Initial multicast
 - Gossip-based FEC-based repairs (proactive)
 - Explicit requests (reactive)



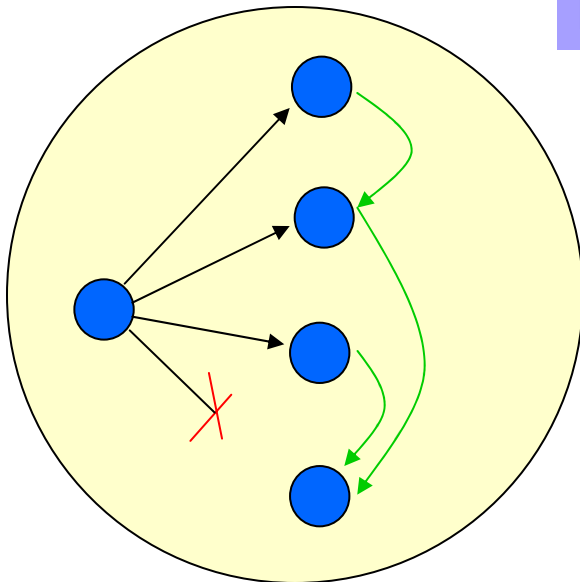


Slingshot: Receiver Based FEC

Slingshot uses **Receiver-Based FEC**:

Senders send initially via unreliable IP Multicast

Receivers repair losses by proactively sending each other FEC repair packets



Each receiver sends an error correction packet to c randomly selected receivers with the last r packets it received

Rate-of-fire parameter (r, c) : Allows tuning of overhead-timeliness tradeoff



Slingshot: Road Map

- Baseline Evaluation:
 - Time-Criticality of Scalable Reliable Multicast measured in single and multiple groups
- Single Group Slingshot: 2 orders of magnitude improvement over baseline (SRM) in a single group
- Multiple Group Slingshot: 20 times improvement over baseline (SRM) in multiple groups
- Vision: Time-Critical Scalable Services Architecture: Allows lightweight services with specific reliability/consistency and timing requirements to be deployed in datacenters, using Slingshot as a core primitive



Slingshot Evaluation Setup

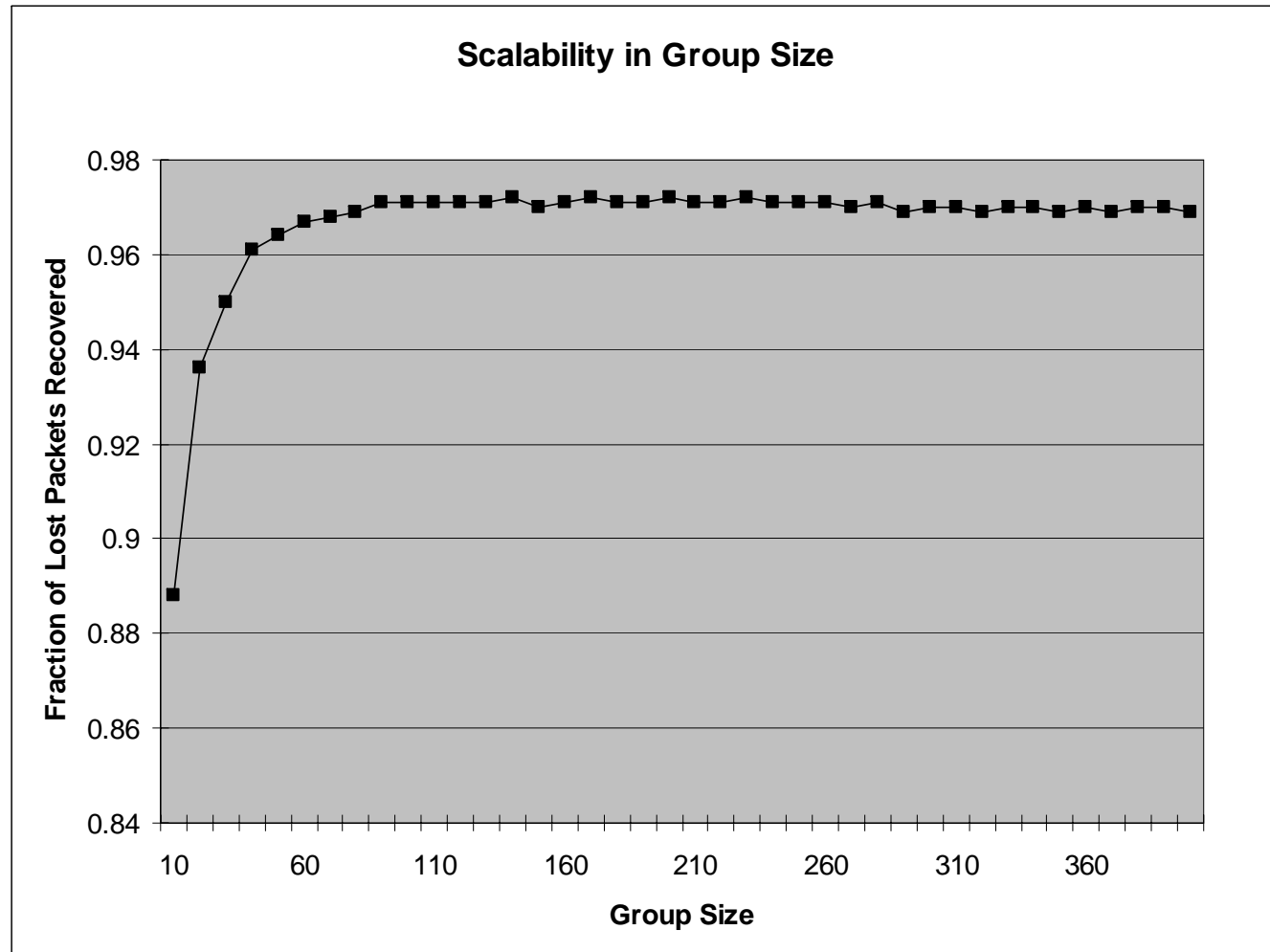
- 64 node rack-style cluster at Cornell. Simulations used for testing scalability to >64 nodes.
- Loss rate fixed at 1%: packets dropped at end buffers
- All nodes send and receive
- Inter-node latencies = 50-100 microseconds
- Single Multicast Group: 1000 packets per second
- Each node multicasts 64 packets per second; i.e one packet every 64 milliseconds



Slingshot Scalability: Group Size

Simulation Results:

Gossip-Style Scalability:
Insensitive to scale beyond a certain size

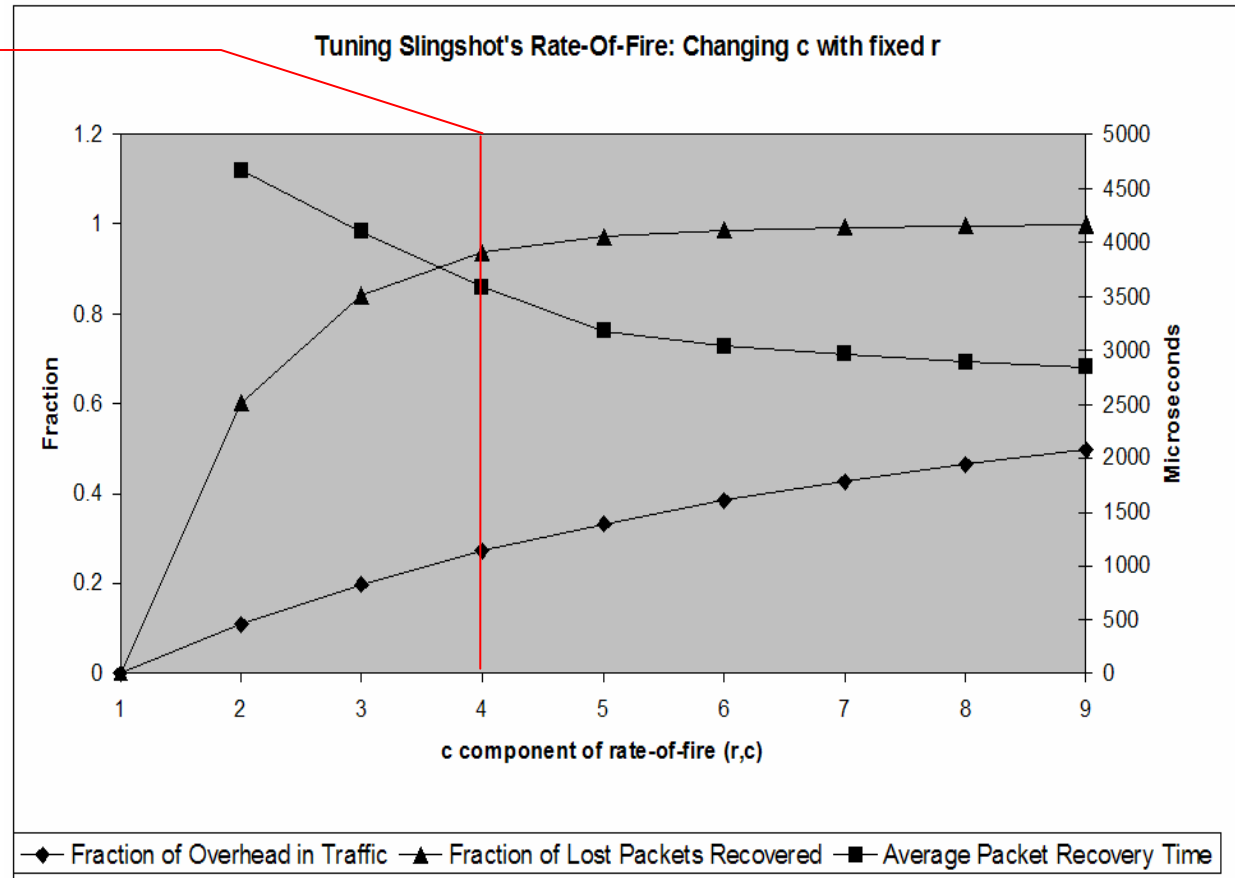




Slingshot Tunability

For 27% overhead, 93.5% Lost Packets are recovered at an avg. of 3.5 milliseconds

Example Tradeoff Points between Overhead, Timeliness, and Reliability →



Overhead and Recovered Packets plotted on left y-axis, Recovery Time on right

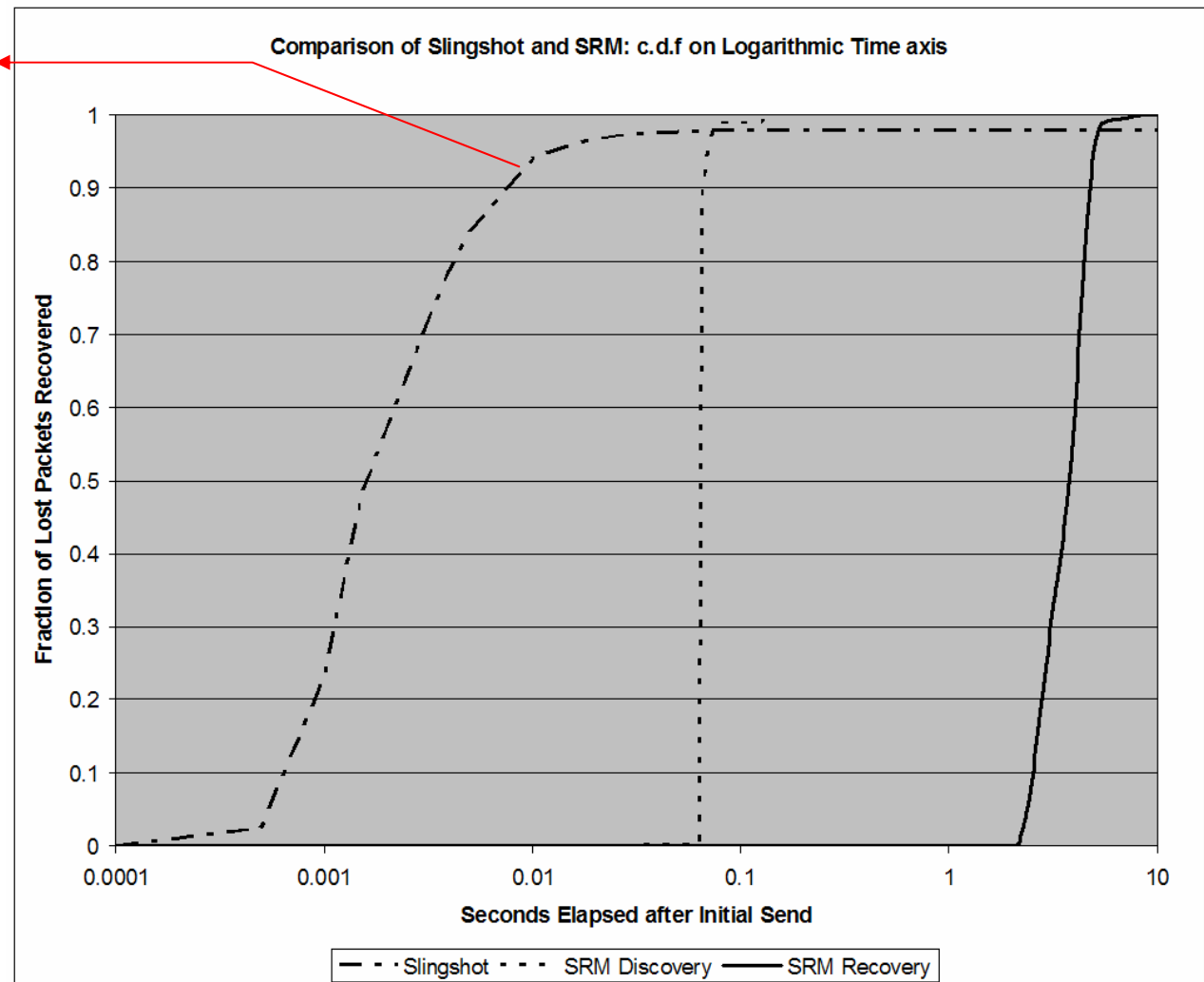


Slingshot vs SRM: One Group

Slingshot recovers 93% in 10 ms, 97% in 25 ms

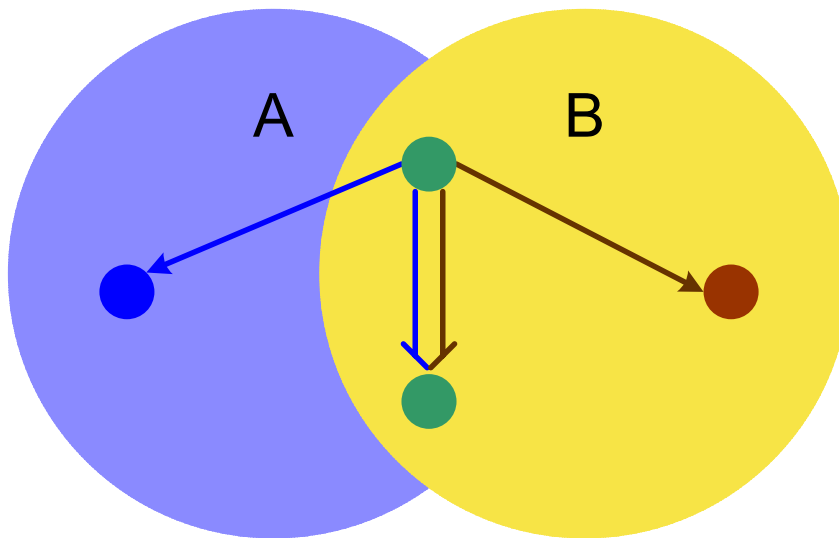
Fastest SRM packet Recovery is 2.2 seconds!
93% in 4.85 seconds,
97% in 5.1 seconds

2-3 Orders of Magnitude Faster





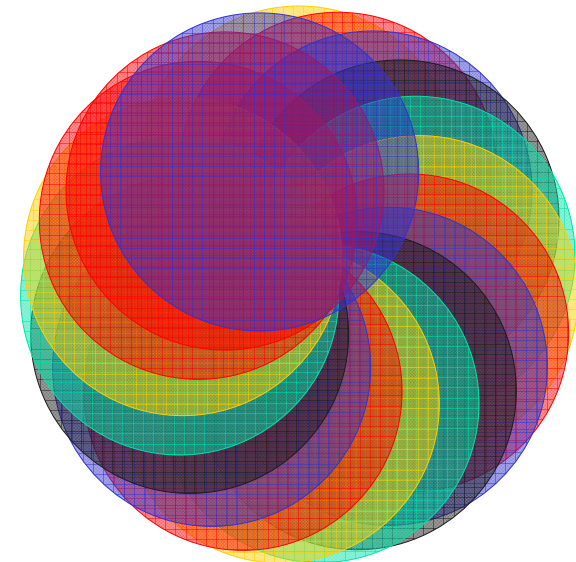
Slingshot in Multiple Groups



$n1$ can send a repair to $n2$ after receiving r packets in either A or B: recovery in the intersection is twice as fast

In intersections of k groups, kx speedup

Arbitrary Overlap Patterns



Extension of this simple concept to multiple groups involves a modification of random target selection, so that each node on an average receives the same number of repairs capable of restoring a given data packet.



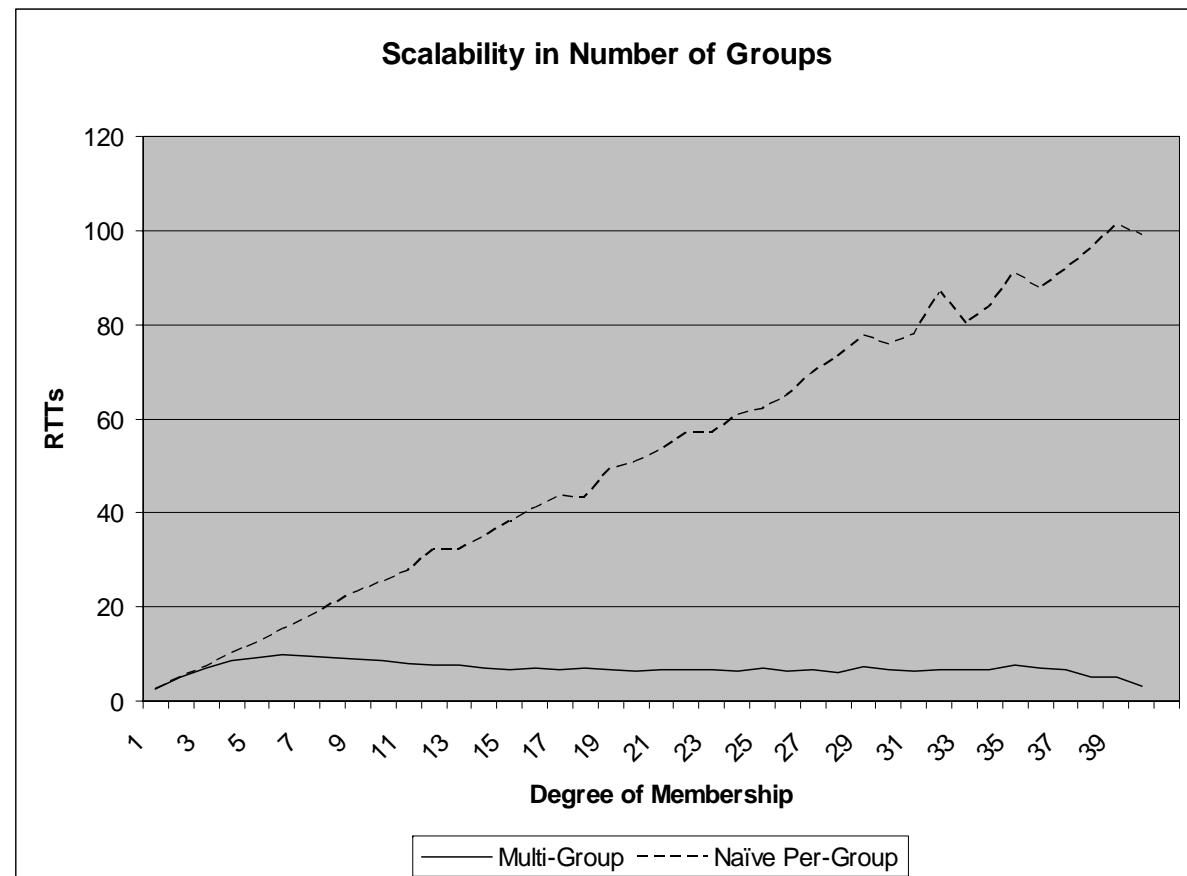
Slingshot Scalability: # of Groups

Simulation Results:

400 Nodes, each node joins x randomly chosen groups, where x is degree of Membership

1 RTT ~ 400 microseconds

Implementation of Multi-Group protocol completed, evaluation in progress



Slingshot Recovery Time is insensitive to # of groups



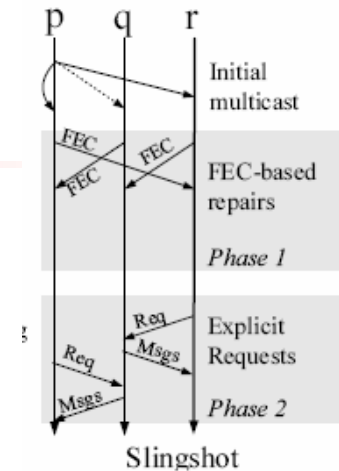
Slingshot Summary

Main idea:

- Proactive gossip of FEC packets before reactive re-transmission

Status:

- Single group:
 - 2-3 orders of magnitude latency reduction over SRM (real implementation)
 - Scalability in group size, resilience to lost packets
- Multiple groups:
 - About factor of 20 in latency reduction over baseline (simulation)
 - Full evaluation in progress





Talk Outline

- Technical approach
- Components
 - Scalable reliable multicast
 - Scalable event processing
 - Large-scale, real-time delivery
- Summary and outlook



The Model

- Event is a set of (attribute, value) pairs
 - Example: Event notifying location of a vehicle
 - $\{(Type, "Tank"), (Latitude, 10), (Longitude, 25)\}$
- Subscription is a set of predicates on event attributes (conjunctive semantics)
 - Example: Subscription looking for tanks in the area
 - $\{(Type = "Tank"), (8 < Latitude < 12)\}$
 - Equality and range predicates



The Problem

- Given: A (large) set of subscriptions, S , and a stream of events, E
- Find: For each event e in E , determine the set of subscriptions whose predicates are satisfied by e
- Scalability:
 - With the event rate
 - With the number of subscriptions



What About State?

- Event is a set of (attribute, value) pairs
 - Example: Event notifying location of a vehicle
 - $\{(Type, "Tank"), (Latitude, 10), (Longitude, 25)\}$
- Subscription is a query over **sequences of events**
 - Example: Subscription looking for adversaries with suspicious behavior
 - "Notify me if enemy first visits location A and then location B"
 - Subscriptions need to maintain state across events



The Problem (Stateful)

- Given: A (large) set of **stateful** subscriptions, S , and a stream of events, E
- Find: For each event e in E , determine set of subscriptions whose predicates are satisfied by e



Stateful Pub/Sub

- Running example: Stock queries
- Example subscriptions (queries):
 - Notify me if MSFT stock price **first** goes up for at least 30 minutes, **then** goes down.
 - Notify me if there is a sale of IBM at price p , and the next sale of IBM has a price above $1.05p$.
 - Any stock increases **monotonically** for at least 30 minutes.



Talk Outline

- Technical approach
- Components
 - Scalable reliable multicast
 - Scalable event processing
 - Stateful publish-subscribe
 - Our approach
 - Architecture and experiments
 - Large-scale, real-time delivery
- Summary and outlook



Stateful Queries

- Novel subscription algebra

- π_X ■ Projection operator

- ρ_f ■ Renaming operator

- σ ■ Selection operator

- \cup ■ Union operator

- $:$ ■ Unbounded sequencing operator

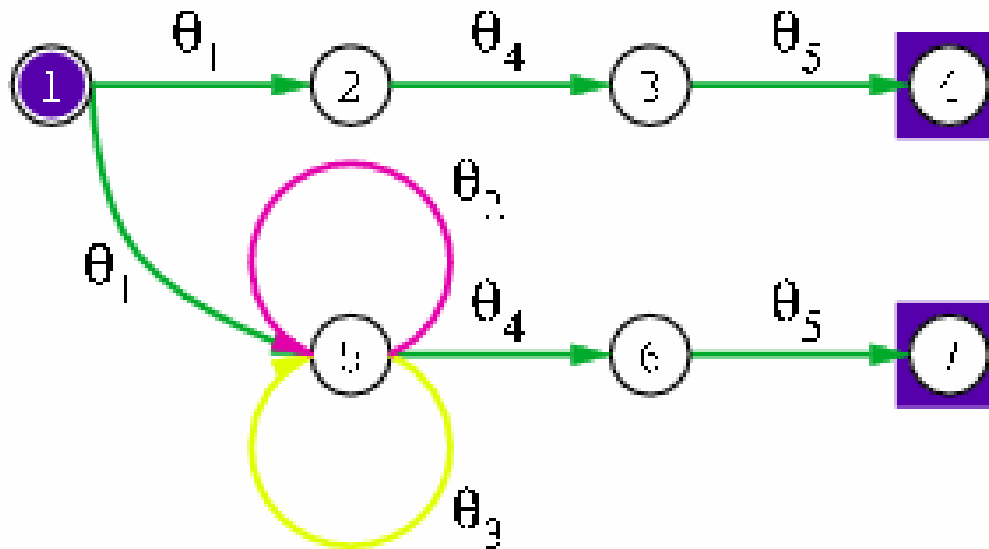
- $;\theta$ ■ Conditional successive sequencing operator

- μ ■ Iteration operator



Managing State

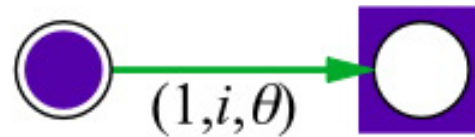
- Use linear finite state automaton with self loops to encapsulate state





Operators

- Relational unary operators (on non-temporal attributes)
 - Selection σ_θ
 - Projection π_X
- Together these give conventional pub/sub

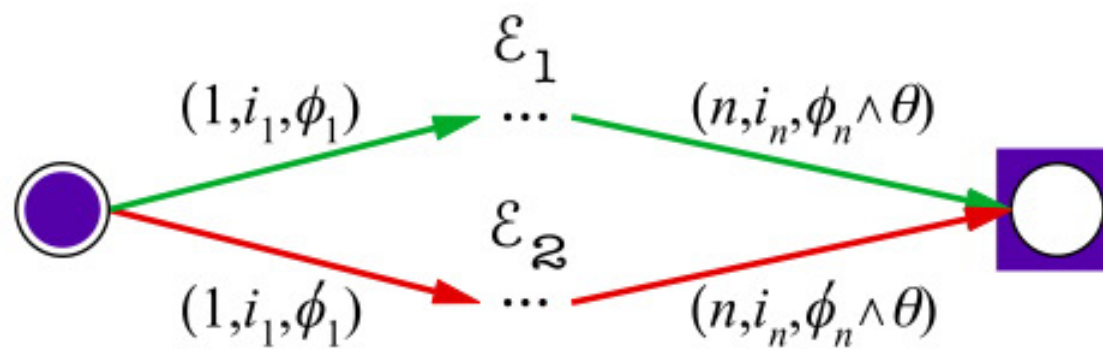


Automaton for $\sigma_\theta(S)$



Operators

- Relational binary operator $S_1 \cup S_2$



Automaton for $\sigma_\theta(\mathcal{E}_1 \cup \mathcal{E}_2)$



Sequence Operator $S_1 ;_{\theta} S_2$

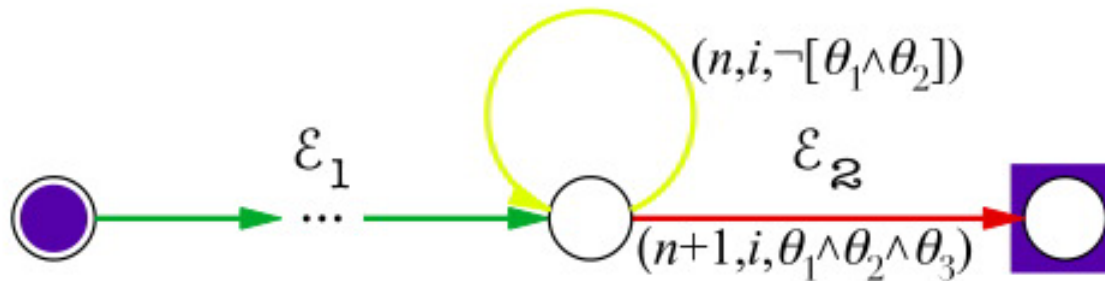
- *Conditional sequence* operator
- Definition:

$$\left\{ \langle \bar{a}_1, \bar{a}_2, t_0^0, t_1^1 \rangle \models \theta \mid \begin{array}{l} \langle \bar{a}_i, t_0^i, t_1^i \rangle \in S_i, t_1^0 < t_0^1, \text{ and} \\ \nexists \langle \bar{b}, s_0, s_1 \rangle \in S_2 \text{ w/ } t_1^0 < s_0, \\ s_1 < t_0^1, \langle \bar{a}, \bar{c}, t_0^0, s_1 \rangle \models \theta \end{array} \right\}$$

- After an event from S_1 , match the first event from S_2 that satisfies the condition
 - No overlap of time intervals



Automaton



Automaton for $\sigma_{\theta_3}(\mathcal{E}_1; \theta_2 \mathcal{E}_2)$

$$\mathcal{E}_2 = \sigma_{\theta_1}(S_i)$$



Iteration Operator (Fixpoint)

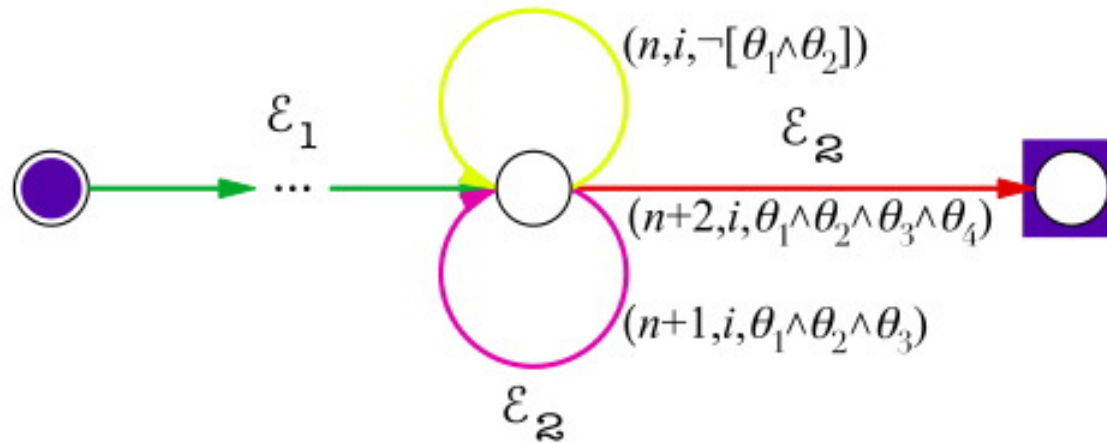
- *Iteration* operator (similar to Kleene-*) $\mu_{\mathfrak{F},\theta}(S_1, S_2)$
- Definition: $\mu_{\mathfrak{F},\theta}(S_1, S_2) = \bigcup_{n \geq 1} \mathcal{S}^{[n]}$

$$\mathcal{S}^{[0]} = \left\{ \langle \bar{a}, \bar{b}, t_0, t_1 \rangle \models \theta \mid \begin{array}{l} \langle \bar{a}, t_0, t_1 \rangle \in S_1, \bar{b} \subseteq \bar{a} \\ \bar{b} \text{ have the attributes in } S_2 \end{array} \right\}$$

$$\mathcal{S}^{[n+1]} = \pi_{S_1 \cup S_2} \circ \mathfrak{F}_{\theta}((\mathcal{S}^{[n]}); S_2)$$



Automaton



Automaton for $\sigma_{\theta_4}(\mu_{\sigma_{\theta_3}, \theta_2}(\mathcal{E}_1, \mathcal{E}_2))$
 $\mathcal{E}_2 = \sigma_{\theta_1}(S_i)$



Aggregates

- Attribute introduction function (a new unary operator)

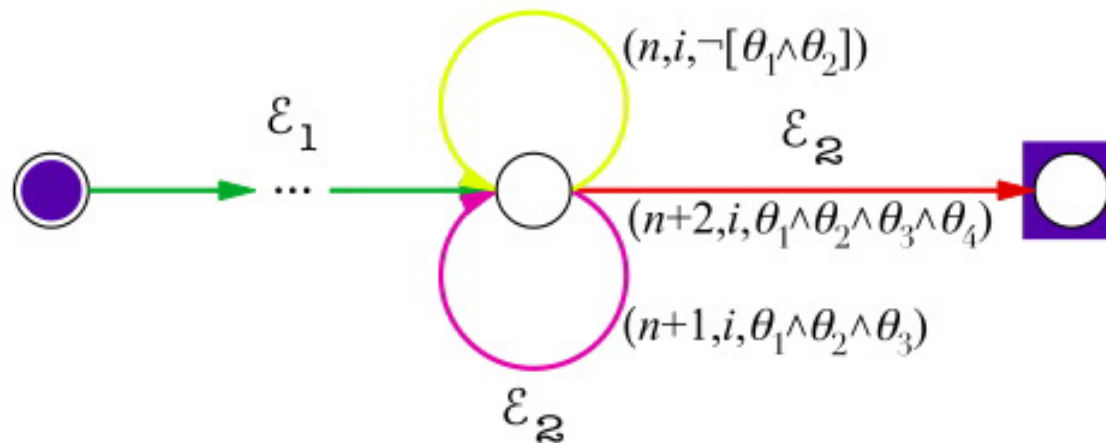
$$\alpha_f(\langle \bar{a}, t_0, t_1 \rangle) = \langle \bar{a}, f(\bar{a}, t_0, t_1), t_0, t_1 \rangle$$

- Computes a term and appends it to the event tuple



Aggregates

- Typical use: in the \mathcal{F} part of an iteration $\mu_{\mathcal{F},\theta}(S_1, S_2)$



- Cannot aggregate over a set of simultaneously-arriving events



Example Query

- Only consider variables with value > 10000 . Notify me when for any such variable, there is a monotonic decrease in its value for at least 10 minutes. Then next time I get an update on this value it is 50% above the previously seen value.
- Intuitive: Starts out large, followed by drop, followed by sudden upwards move
- Expressive in general for large class of monitoring queries ("**systemic monitoring**")



Talk Outline

- Technical approach
- Components
 - Scalable reliable multicast
 - Scalable event processing
 - Stateful publish-subscribe
 - Our approach
 - Architecture and experiments
 - Large-scale, real-time delivery
- Summary and outlook



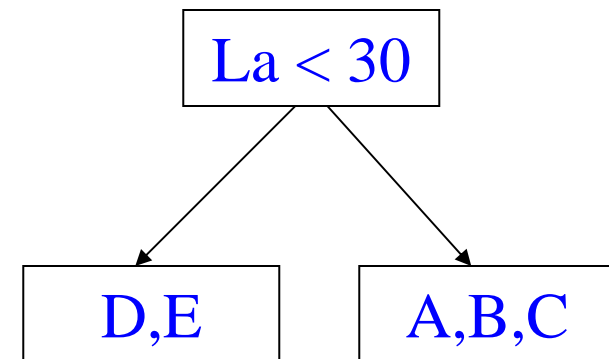
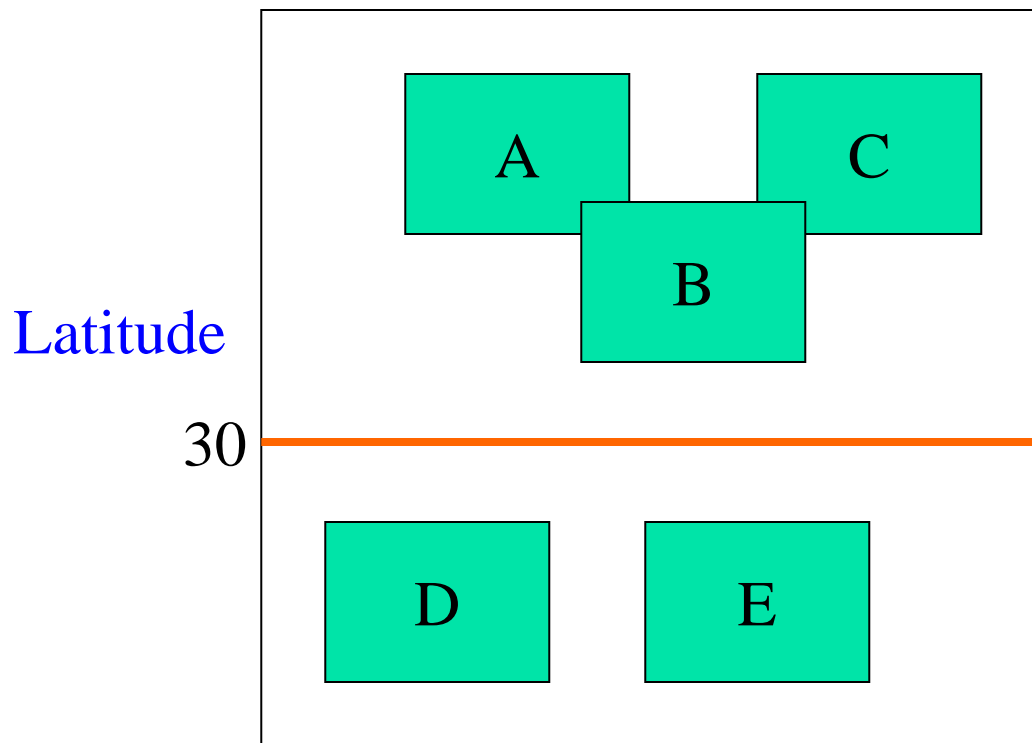
Which Baseline?

- Possibility 1: Implementation as triggers in today's database systems
 - Triggers scale very poorly: Each trigger predicate is evaluated for each insert (INSERT triggers on a table)
- Possibility 2: Implementation only at the app-server level (J2EE with custom data structures)
 - No persistence, lots of custom code, poor performance
- Possibility 3: Implementation using pub/sub system with custom component at the app server
 - Our baseline scenario



Basic Pub/Sub

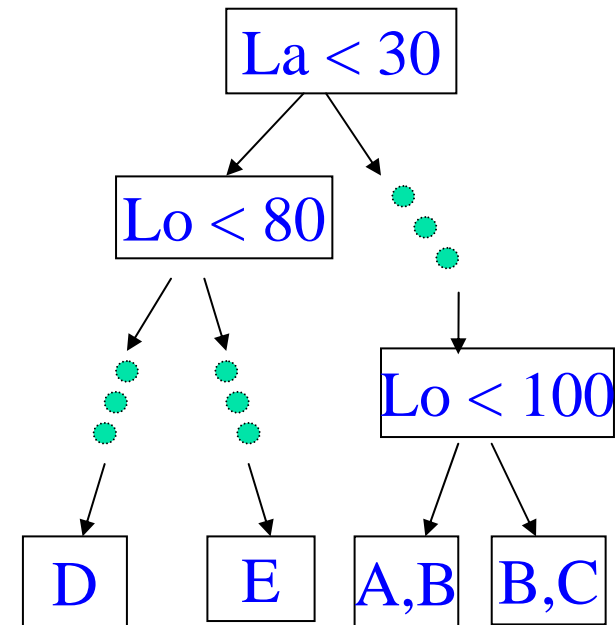
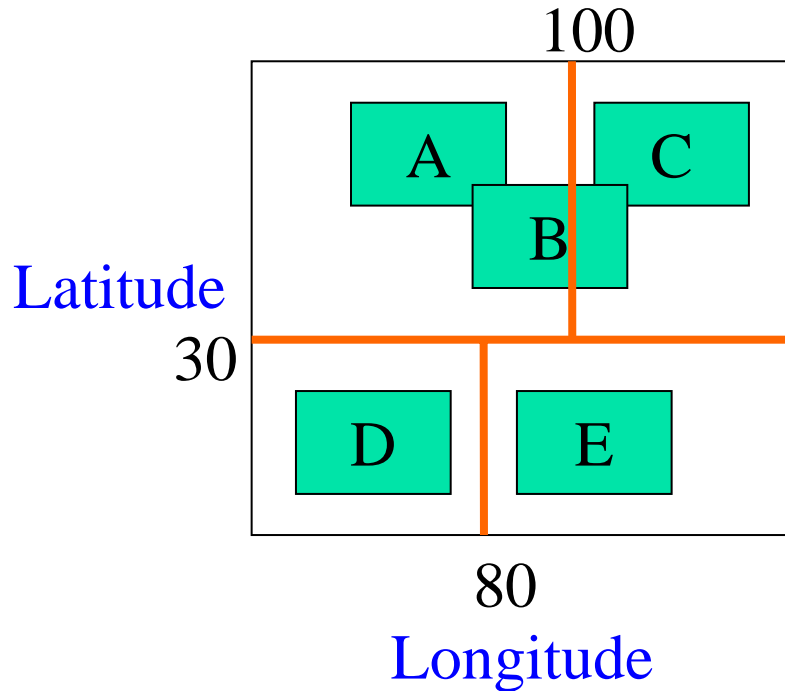
- Indexing subscriptions





Basic Pub/Sub

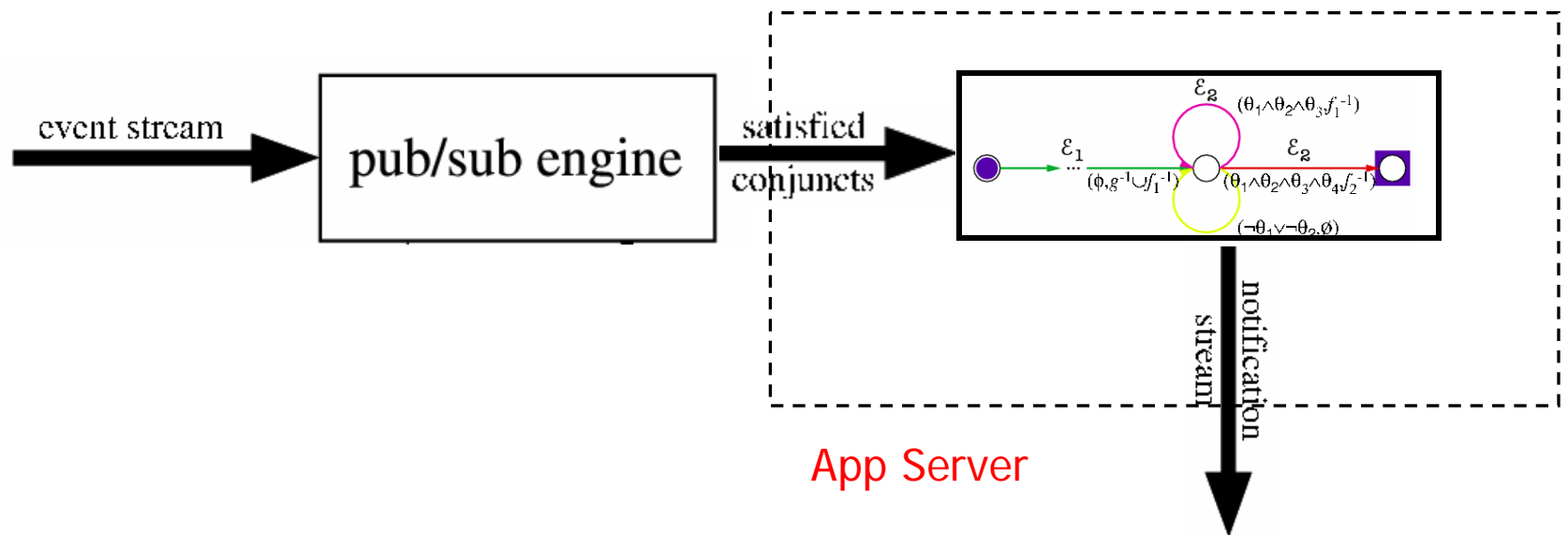
- Indexing subscriptions





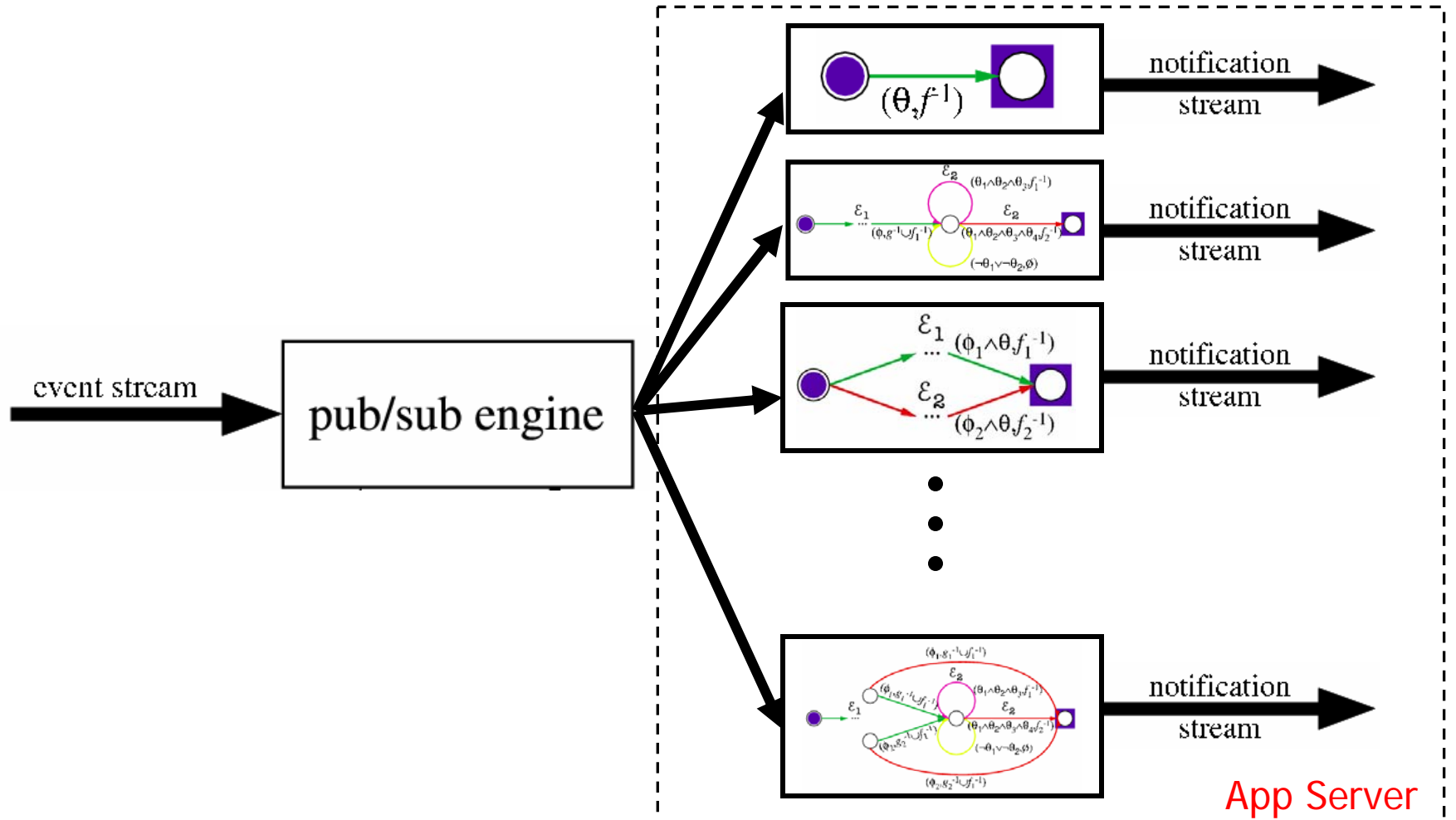
Baseline System Architecture

- Pub/sub engine with state management at the application server



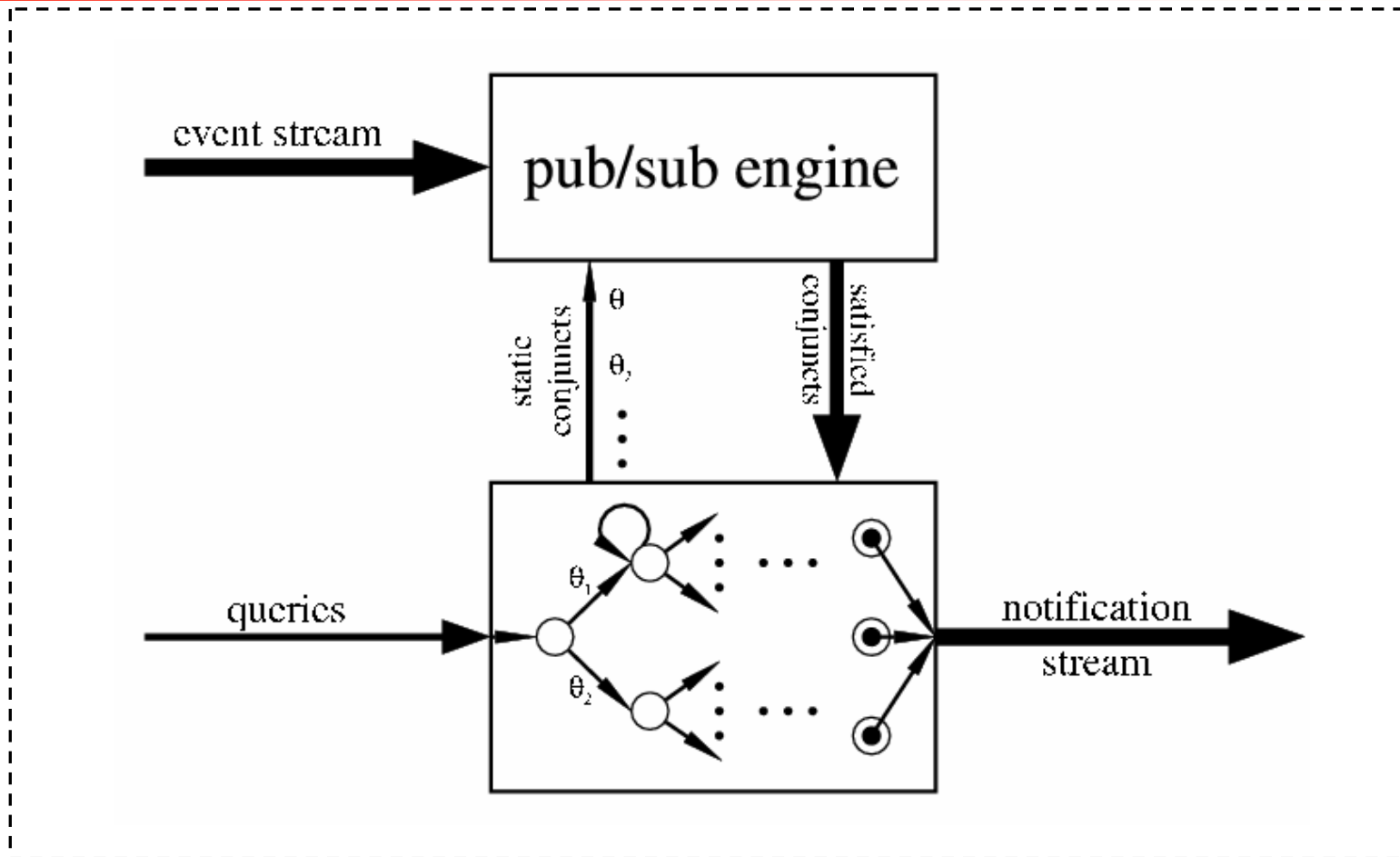


System Architecture (Contd.)





Our Approach





Experimental Setup

- Compares baseline to Cayuga System
 - Technical benchmark
 - (Application benchmark)
- Components
 - Subscription Generator
 - Event Generator
 - Pub/Sub Engine (Le Subscribe/RPH)
 - State Manager



Experimental Setup

Index Building Parameters

- Fill factor of RPlus Tree (8)
- Max number of discrete attributes (16)
- Max number of continuous attributes (16)

Hardware:

- 2.2 GHz Pentium 4 processor with 1GB of main memory



Event Generation Parameters

- Fraction of discrete attributes (0.5)
- Number of events (100,000)
- Number of attributes per event (8)
- Discrete attributes
 - Zipfian for discrete attributes (0.2)
 - Domain size for discrete attributes (100)
 - Zipf for attribute value generation (0.2)
- Continuous attributes
 - Zipfian for continuous attributes (0.2)
 - Stddev (0.25)



Subscription Generation Parameters

- Number of subscriptions (100,000)
- Fraction of discrete attributes (0.5)
- Mean number of attributes per subscription (4)
- Discrete attributes:
 - Zipfian for discrete attributes (0.2)
 - Domain size for discrete attributes (100)
 - Zipf for attribute value generation (0.2)
- Continuous attributes:
 - Zipfian for continuous attributes (0.2)



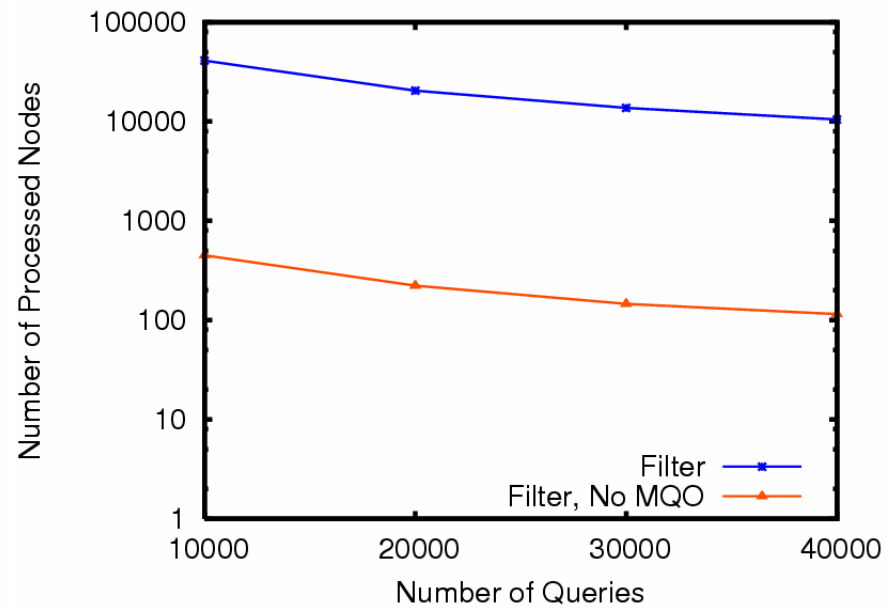
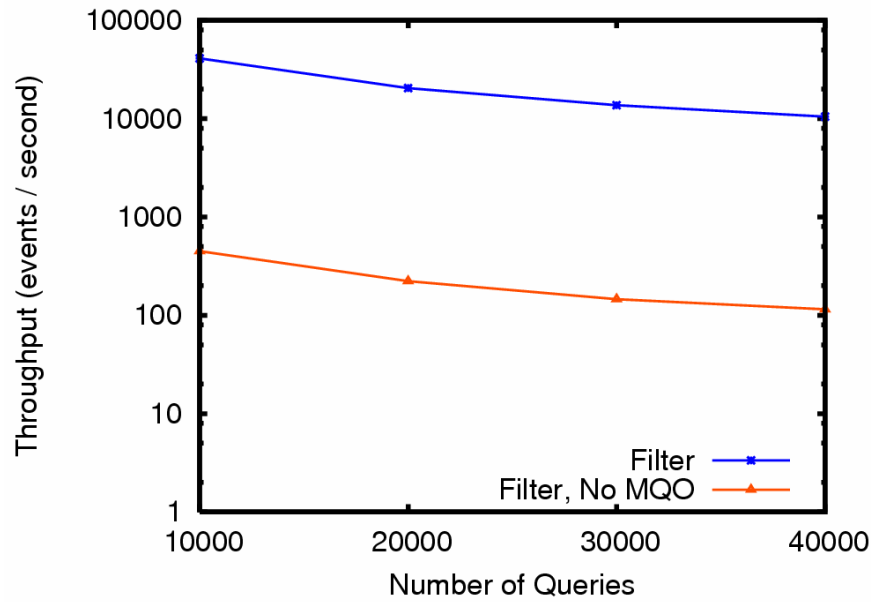
Default Values of Parameters

Variable	Value
$ n_e $ (Number of events in the stream)	10000
# number of predicates per event	8
$ n_a $ (Number of different attributes (discrete + continuous))	4+4
$ n_{ss} $ (#Sequence queries)	20000
c_s (#predicates per Step (equality + inequality))	2 + 2
equality predicate value domain	100
number of range choices for an atomic inequality predicate	25
selectivity of atomic inequality predicate	0.7
zipf parameter	1
c_m (#steps per Sequence)	3
duration of a sequence subscription	≤ 20
# iterations	2



Experimental Results

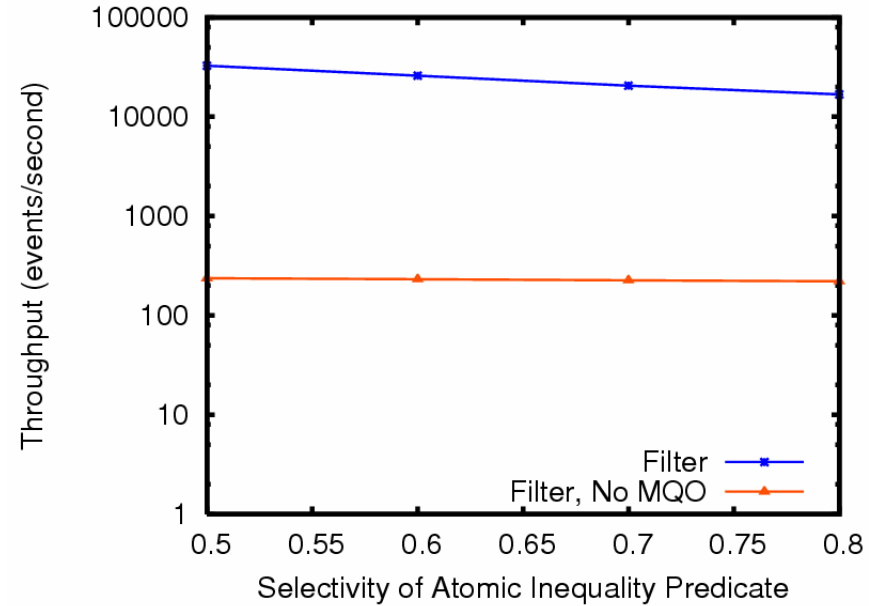
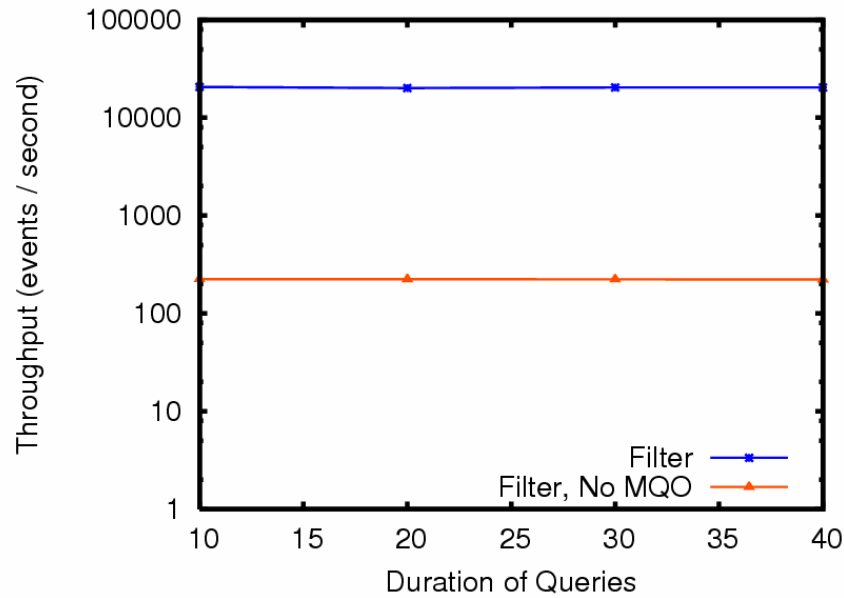
Y axis in Log scale!





Experimental Results (Contd.)

Y axis in Log scale!





Cayuga: Summary

Main ideas:

- Stateful publish-subscribe, novel algebra, mapping to linear FSA, efficient indexing

Status:

- 2-3 orders of increase in throughput over baseline (real implementation)
- Full evaluation and extensions in progress



Talk Outline

- Technical approach
- Components
 - Scalable reliable multicast
 - Scalable event processing
 - Large-scale, real-time delivery
- Summary and outlook



ChunkySpread

- End-system wide-area multicast
 - End-hosts handle packet replication and routing
- Replacement for wide-area IP multicast
 - Solves IP multicast scaling issues
 - Adds functionality (i.e., buffering, congestion control)



ChunkySpread Innovations

- 1998: End-system multicast (ESM) invented by Francis (Yoid)
 - Single-tree: heavy load at branching nodes, no useful work from leaf nodes
- 2002: Multi-tree ESM invented (SplitStream)
 - Based on a structured DHT underlay: complex, uniform workload across hosts (rigid)
- ***ChunkySpread breakthrough***: Simple unstructured underlay, fine-grained control over workload



ChunkySpread Technology

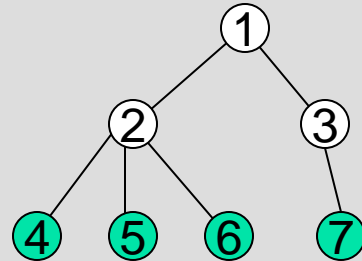
- Based on underlying random graph
 - Formed through simple random walks
 - Node degree proportional to individual node's desired load
 - Unique capability
- Each node locally negotiates parent-child relationship with neighbors for each tree, considering:
 - Load
 - Path length
 - Tit-for-tat (prevents DDoS attacks)



Technology Comparison

Baselines

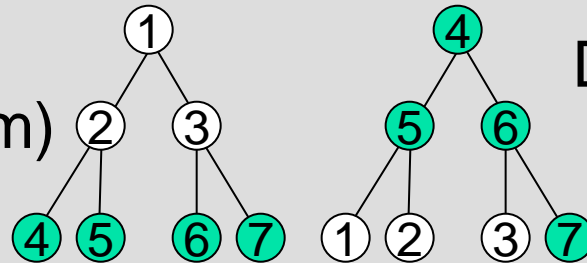
Single Tree
(Yoid, ESM)
1998



Bottlenecks

Do no useful work

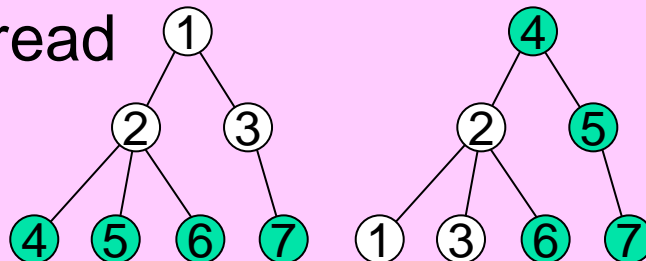
Multi-Tree
(Splitstream)
2002



DHT-based: complex

Rigid, uniform load

ChunkySpread
2004



Unstructured: simple

Fine-grained load control



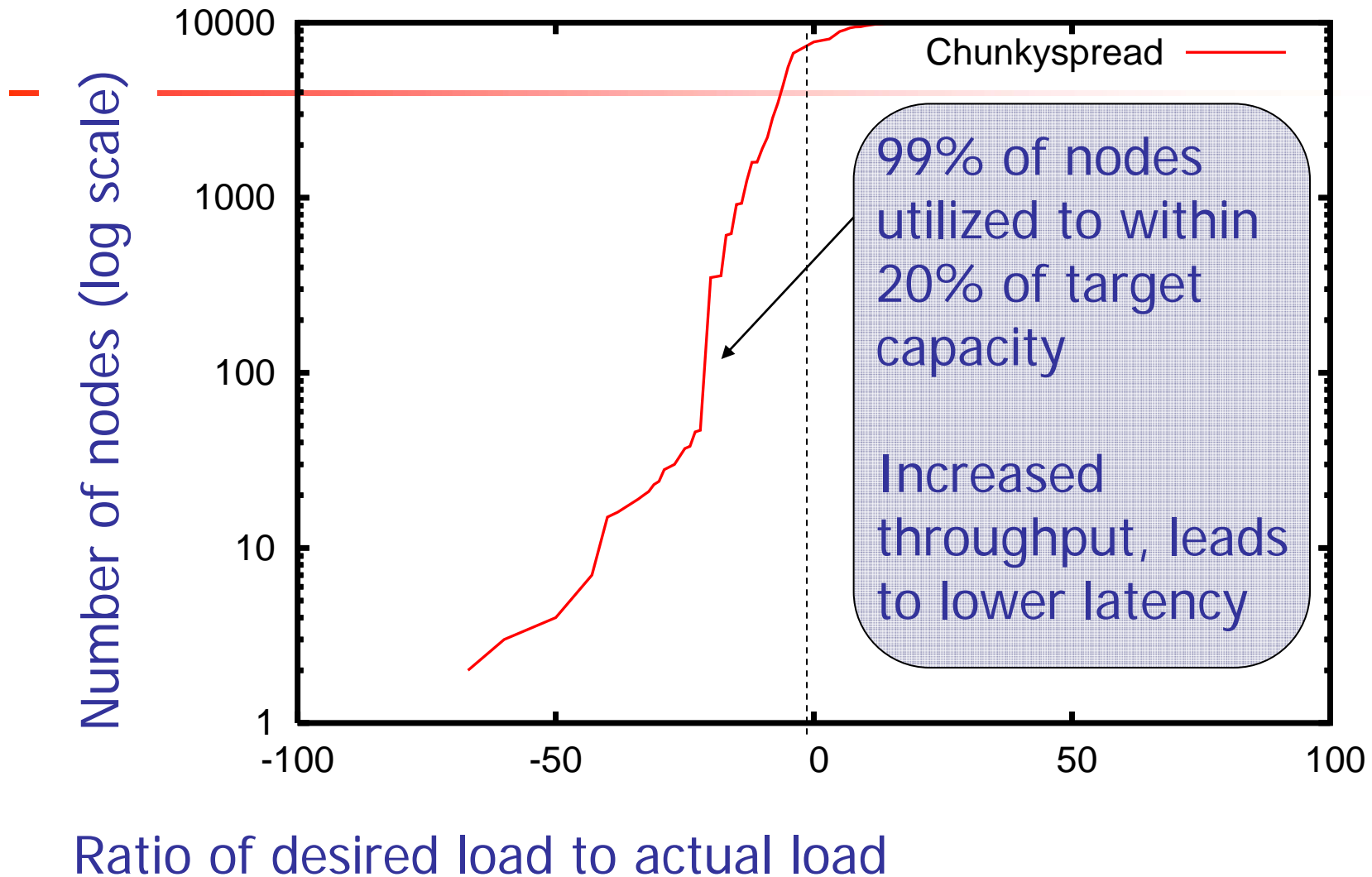
Throughput Comparison

	Homogeneous node capacities (1Mbps baseline)	Heterogeneous 10:1 ratio, uniformly distributed
Single Tree	500Kbps (uniform fanout of 2)	N/A
Split-Stream	1Mbps	1Mbps
Chunky-Spread	1Mbps	~5Mbps*

* Bottleneck is sending capacity



Cumulative distribution of load (for 10:1 heterogeneity ratio)





ChunkySpread Summary

Main idea:

- Decentralized construction of random network graphs with fine-grained control over node degree

Status:

- Factor of five improvement in throughput
- Full implementation in progress



Talk Outline

- Technical approach
- Components
 - Scalable reliable multicast
 - Scalable event processing
 - Large-scale, real-time delivery
- **Summary and outlook**



Status: July 2005

- Scalable reliable multicast: Slingshot
 - One group: > 2-3 orders of magnitude latency reduction (real implementation)
 - Many groups: > 20 times latency reduction (simulation)
- Scalable event processing: Cayuga
 - Event processing: 2-3 orders of magnitude higher throughput (real implementation)
- Large-scale, real-time delivery: ChunkySpread
 - Factor of 5 increase in throughput (simulation)



Outlook

- Finish components and evaluation
- Integration into the Reflective Self-Regenerative Systems (RSRS) architecture
- Interesting open problems
 - Reflection
 - Resubscription