failure notifications never fail… or do they?

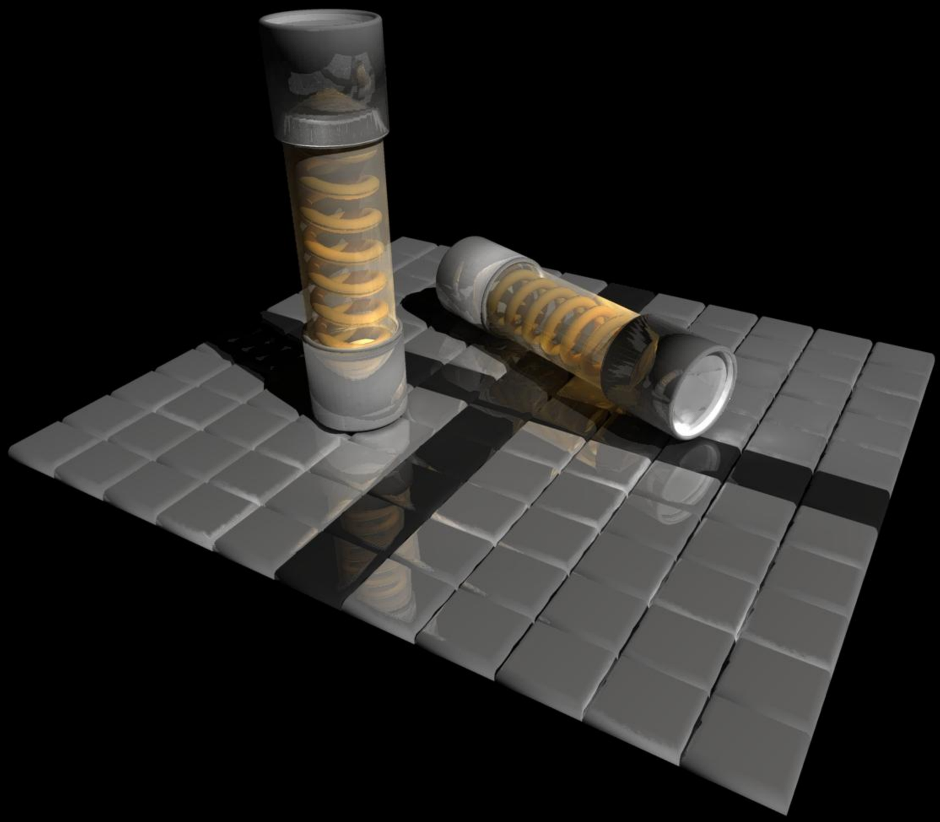
FUSE: LIGHTWEIGHT GUARANTEED DISTRIBUTED FAILURE NOTIFICATION

Paper By:

John Dunagan
Nicholas J.A. Harvey
Michael B. Jones
Dejan Kostic
Marvin Theimer
Alec Wolman

Presentation by:

Rahul Potharaju
EECS 345



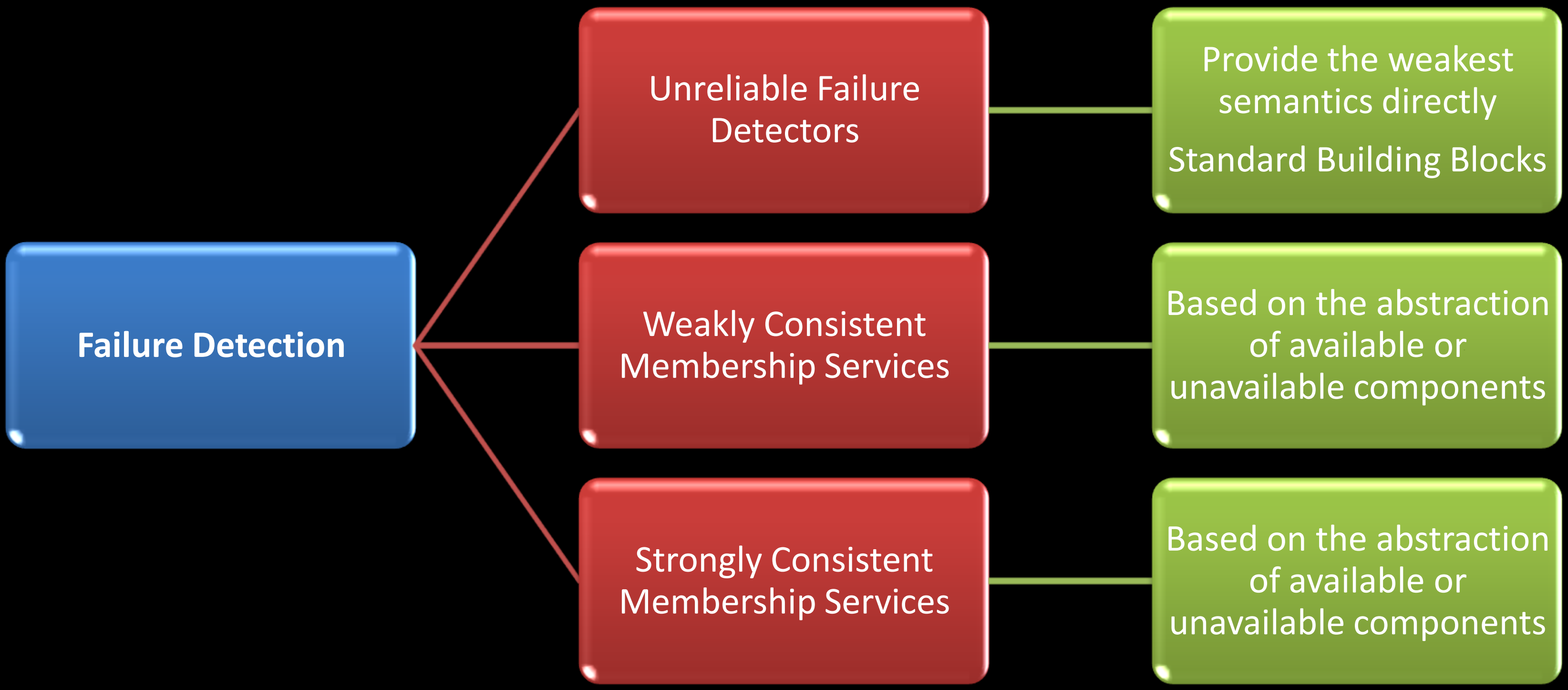
Introduction

- Background
- Overall theme

What is it all about?

- How it works?
- Implementation
- Evaluation

Background…


Failure Detection
Unreliable Failure Detectors
Provide the weakest semantics directly
Standard Building Blocks
Weakly Consistent Membership Services
Based on the abstraction of available or unavailable components
Strongly Consistent Membership Services
Based on the abstraction of available or unavailable components

Background...

Unreliable Failure Detectors

- Introduced by Chandra and Toueg
- Used to solve consensus
  - Alice → Bob → Alice → Bob … It never ends!
- Provide periodic heartbeats saying "I'm alive"
- Provide a semantic guarantee despite the "unreliable" notion: fail-stop crashes will be identified within a bound time – FUSE uses this but provides a stronger version

Weakly Consistent Membership Services

- Lots of work in this field. They differ in speed, accuracy etc...
- FUSE handles intransitive connectivity problems
  - A → B works
  - B → C works
  - A → C doesn't work

Strongly Consistent Membership Services

- Share the abstraction of a membership list too, but they guarantee that all nodes see a consistent list by using atomic updates
- Performs well only at a small scale

FUSE Outline

- Robust programming model that simplifies application development

- Guaranteed failure notification within bounded period of time

- Applications create a FUSE group with an immutable list of participants
    - FUSE monitors this group till a failure is detected by FUSE or application triggers failure.
    - Thus, responsibility of detecting failures shared between FUSE and application.

- Applications: Wide-area internet applications such as content delivery networks, peer-to-peer applications, web-services and grid computing

FUSE Workflow

More in the implementation stage…

Basic Flow:

- Every node in the system runs a FUSE layer.
- Can create multiple FUSE groups between same set of nodes.
- Application invokes the corresponding API to create a FUSE group
  FuseId CreateGroup(NodeId[] set)
- FUSE layer on every node contacted (possibly concurrently) and initialized.
- Application passes on FuseId to every node in the set.
- Each node registers a callback associated with the FuseId using the
  void RegisterFailureHandler(Callback handler, FuseId id)

```
//Creates a FUSE notification group containing the nodes in the set
FuseIdCreateGroup (NodeId[] set)

//Registers a call back function to be invoked when a notification occurs for the FUSE group
Void RegisterFailureHandler (Callbackhandler, FuseId id)

//Allows the application to explicitly cause FUSE failure notification
Void SignalFailure(FuseId id)
```

FUSE Workflow

Success

Success is reported to the creator

Failure

Member unreachable?

Invoke Failure Handlers (Members already informed) → Part of FUSE Garbage collection mechanism

Attempt to associate with a non-existent handler?

Invoke failure handler

What happens subsequently?

- Nodes periodically ping each other.

- If a node initiates a ping that is missed, the node itself stops responding to future pings: ensures that individual observation of a failure converted into a group notification.

- Nodes notified of failure through callback

- Failure notification can be triggered
  - explicitly, by application
  - or implicitly when FUSE detects communication failure among group members.

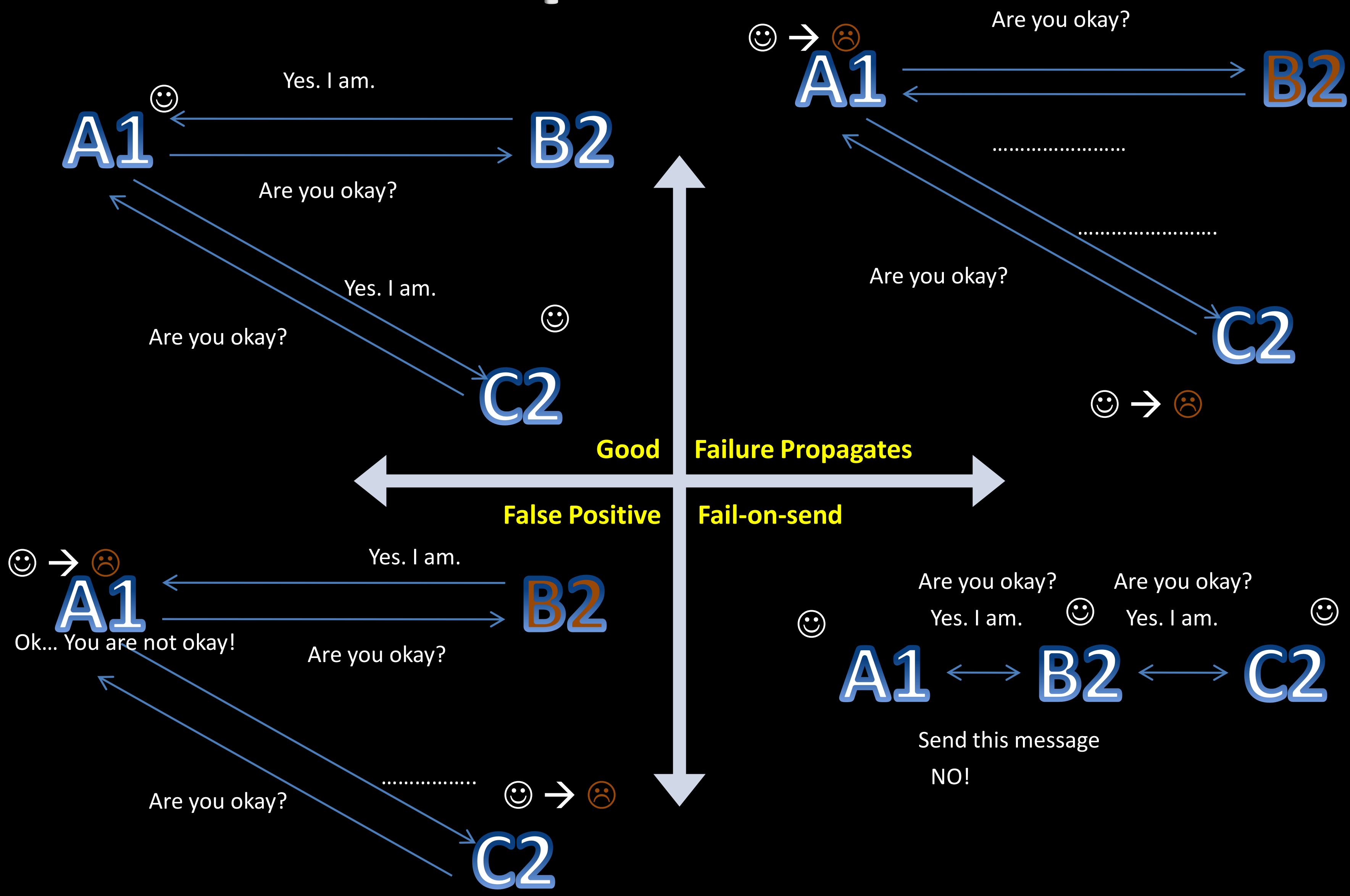
- A node can never know if the failure was caused due to a network failure or a node failure.

- Danger of false positives exist.

- FUSE members on both sides of the partition will receive failure notifications, but it is not possible to communicate additional information across the partition.

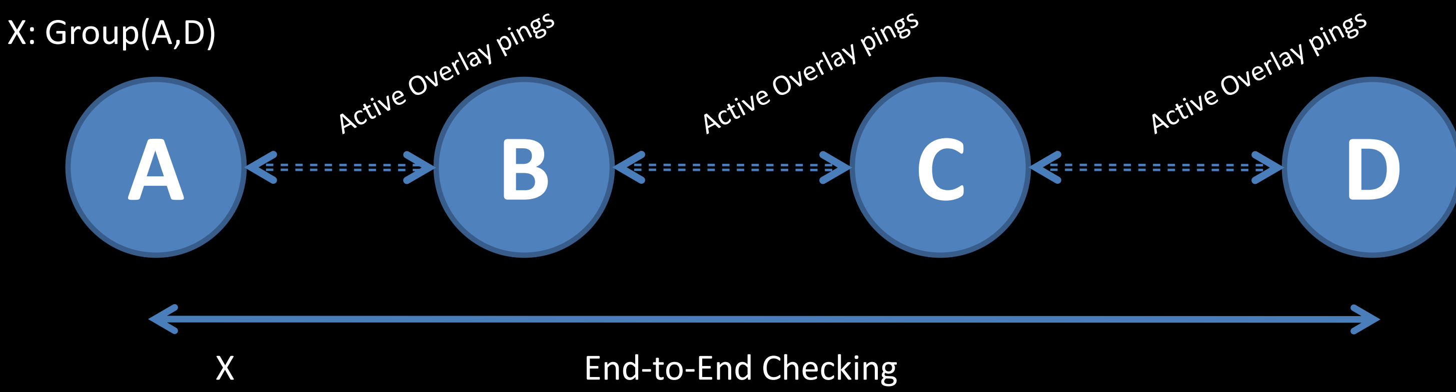
FUSE Principle

B2 lights the fuse and the failure affects the entire group…



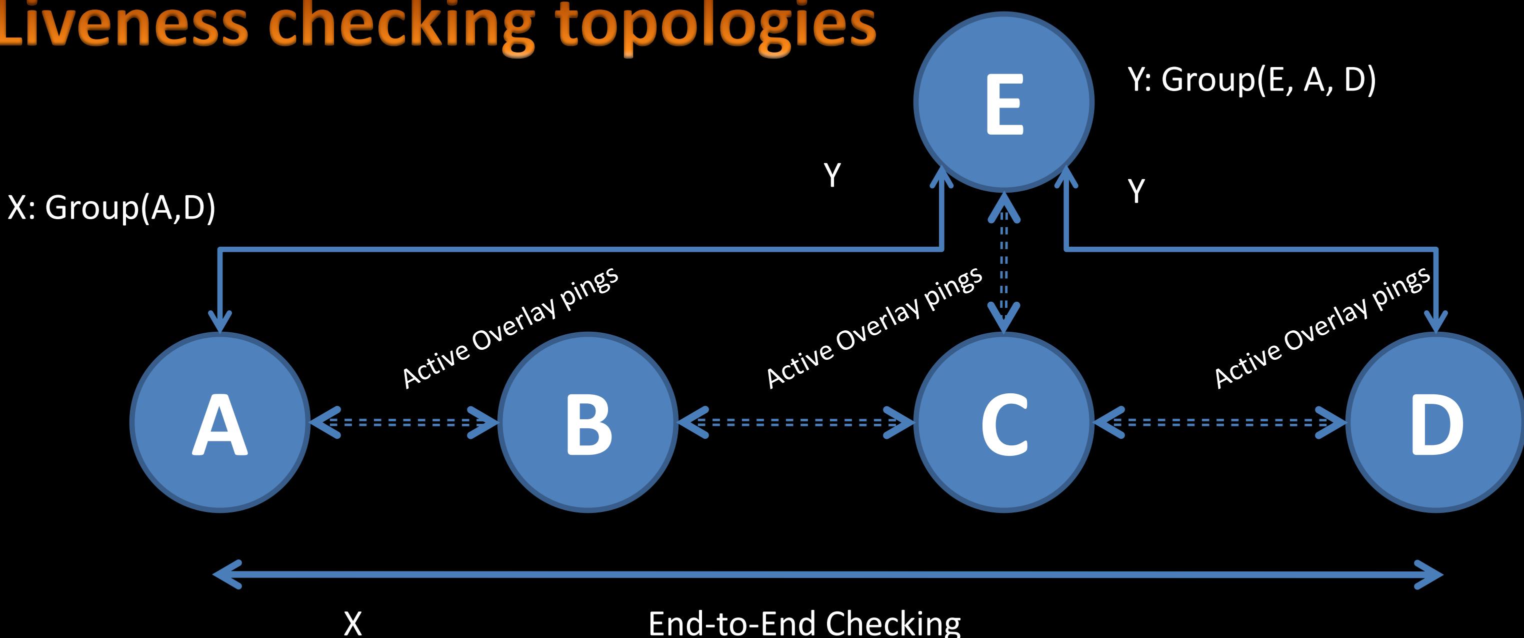
What happens subsequently?

- **Crash recovery:**
  - A recovering node does not know if a failure notification was triggered.
  - FUSE handles this by nodes actively comparing the live FUSE groups during liveness checking.
  - FUSE does not use stable storage, but can be used for masking transient failures.
- **Liveness checking topologies:** per-group spanning trees on an overlay network
  - Constructing liveness topologies on overlay networks allows existing overlay liveness checks to be reused.
  - Overlay nodes that are not part of FUSE groups may not forward failure notifications.



A FUSE group with two members being monitored by overlay pings

Liveness checking topologies



Two FUSE groups being monitored by overlay pings (!(B → C) → (B → A) & (C → D)

Security-Scalability trade-offs

C is a Malicious Node:
Violates the FUSE Principle by not participating

E → C: Can you forward a message to A?
C: Sure. No problem.
…
…
C: I lied to you! I won't forward it.
A never receives a message

C is a Malicious Node:
Violates the FUSE Principle by delaying messages

E → C: Can you forward a message to A?
C: Ok
…
After sometime
C: I almost forgot about E.
A receives a message after sometime

C is a Malicious Node:
Violates the FUSE Principle by initiating DoS Attacks

C → E: Hey… A failed to respond
E: Ok. I'm going down
C → A: Hey… E failed to respond
A: Ok. I'm going down
…
Repeats the same after recovery

Things that could go wrong

Security-Scalability trade-offs

- Violation of FUSE semantics: Dropped notifications
    - handled using multiple dissemination trees
    - Can use all-to-all pinging – but high overhead.
  - By delegates (overlay nodes that are not actually members):
    - use per-group spanning trees without using overlay nodes
    - Increases the amount of liveness checking traffic.
- DoS attacks: malicious node causing frequent unnecessary failure notifications.

Before we go ahead…
(into implementation details)

- Scalable overlay networks such as Chord, CAN, Pastry, and Tapestry have recently emerged as flexible infrastructure for building large peer-to-peer systems.
    - They provide no control over where data is stored
    - No guarantee that routing paths remain within an administrative domain whenever possible
- Meet SKIPNET

SkipNet

- SkipNet is a scalable overlay network that provides controlled data placement and guaranteed routing locality by organizing data primarily by string names

- SkipNet allows for both fine-grained and coarse-grained control over data placement: Content can be placed either on a pre-determined node or distributed uniformly across the nodes of a hierarchical naming sub tree

- An additional useful consequence of SkipNet's locality properties is that partition failures, in which an entire organization disconnects from the rest of the system, can result in two disjoint, but well-connected overlay networks.

- When an entire organization disconnects from the rest of the system, repair of only a few pointers quickly enables efficient routing throughout the disconnected organization; full repair is done as a subsequent background task. These same operations can be later used to efficiently reconnect an organization's SkipNet back into the global one.

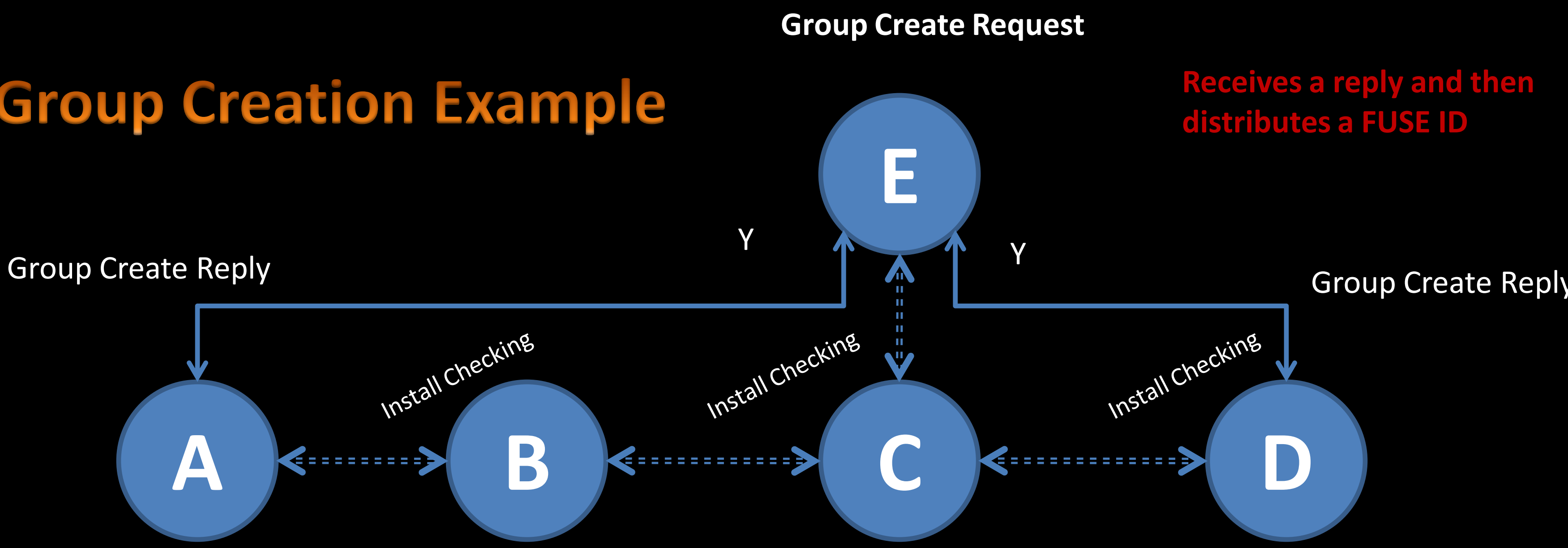
FUSE Implementation

- Implemented on top of SkipNet
- SkipNet features
  - Messages routed through the overlay result in a client up call on every intermediate overlay hop.
  - Overlay routing table is visible to the client.
- Route directly between members during creation and failure notifications – reduces false positives.
- Group creation:
  - Creation request/response directly between root and member nodes
  - Members simultaneously route InstallChecking messages through the overlay towards root. This prepares overlay nodes for future liveness forwarding

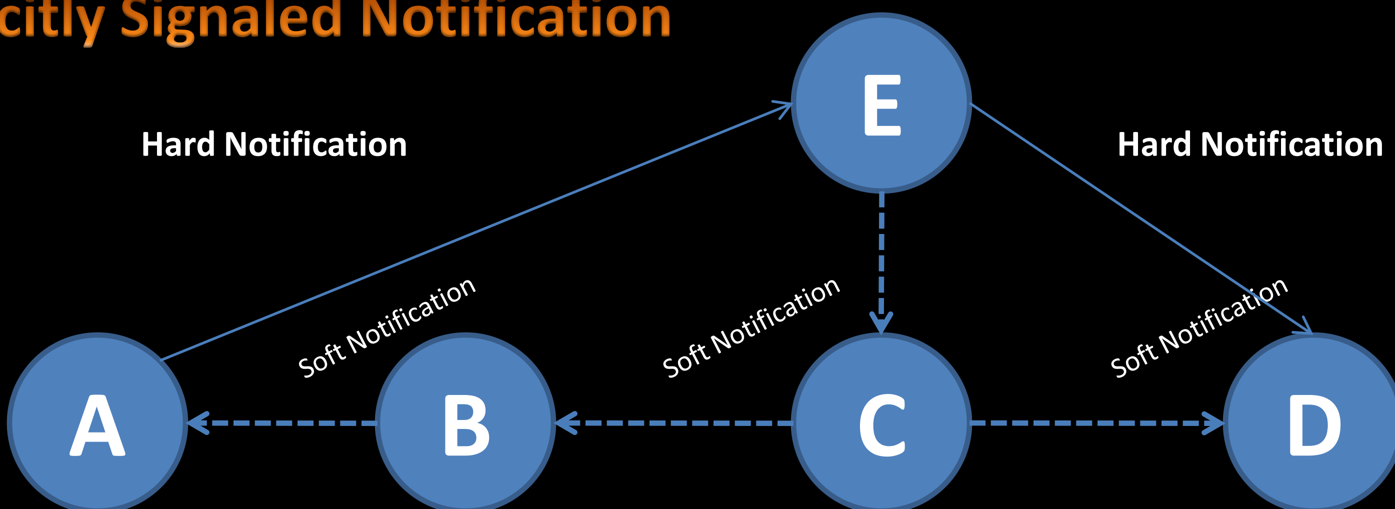
FUSE Implementation

- Steady-State
  - Piggyback a hash containing all FUSE groups that use a particular overlay link on the SkipNet ping messages.
  - Reuse overlay routing table maintenance traffic for liveness checking
- Notifications
  - Hard notifications used to dismantle the group
    - Direct communication. Reduces latency.
  - Soft notifications used to clear state on the liveness checking tree.
    - Member receiving soft notifications initiates repair directly with the root (group creator).
    - Provides resilience to delegate failures.
- Repair
  - NeedRepair msg: Sent by members to root. (In order to reduce latency)
  - SoftNotification: Sent by delegates to root.
  - Otherwise repair mostly similar to group creation.

Group Creation Example



Explicitly Signaled Notification



Experiments

- Latency of group creation: As group size increases, latency increases since although nodes contacted in parallel, probability of encountering a slow link is increased.
  - Note: Groups created by direct messages and hence unaffected by the size of the network.

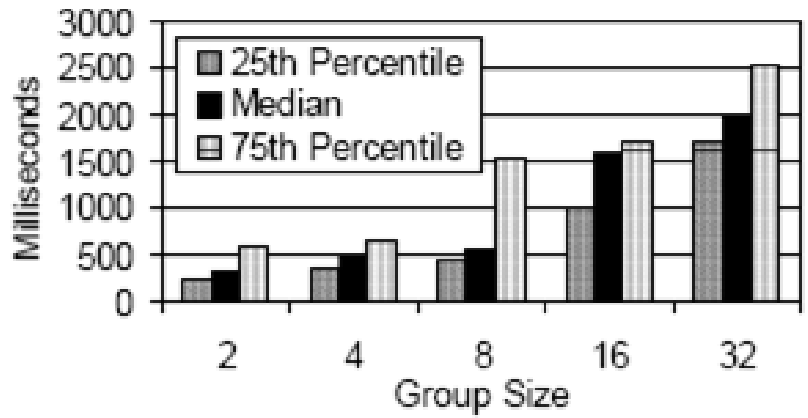


Figure 8. *Latency of Group Creation*

Experiments

- Latency of Failure notification
  - Explicit notification:- Lower than creation due to
    - cached TCP connections
    - One-way message
    - Non-blocking.
  - Crash failures: with ping interval of 1 min and timeout of 30 secs. – TCP connection timeout dominates.

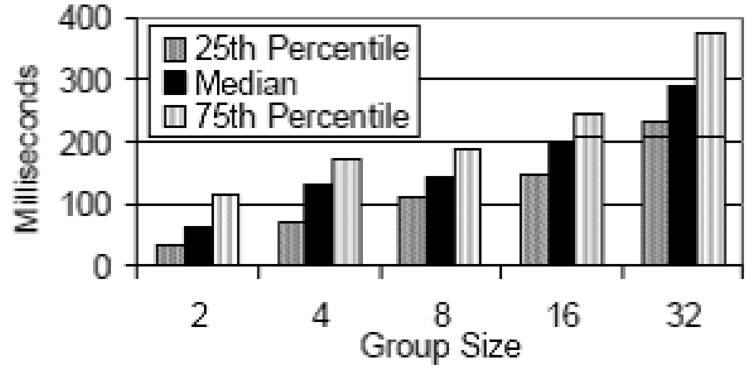


Figure 9. *Latency of Signalled Notification*

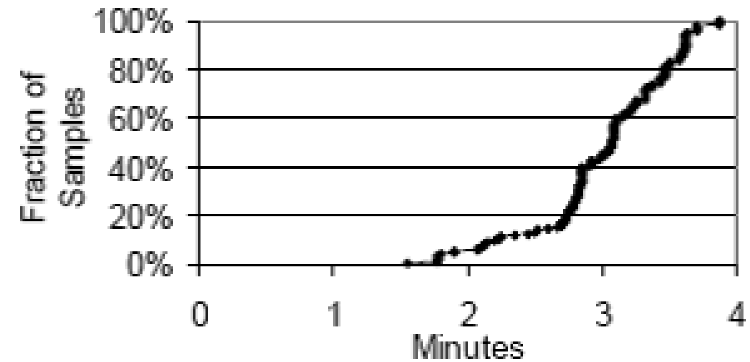


Figure 10. *Combined Latency of Ping Failure, Repair Failure, and Failure Notification. TCP connection timeout during repair dominates other factors.*

Experiments

Significant difference in the latency of group creation and latency of signaled notification – Why?

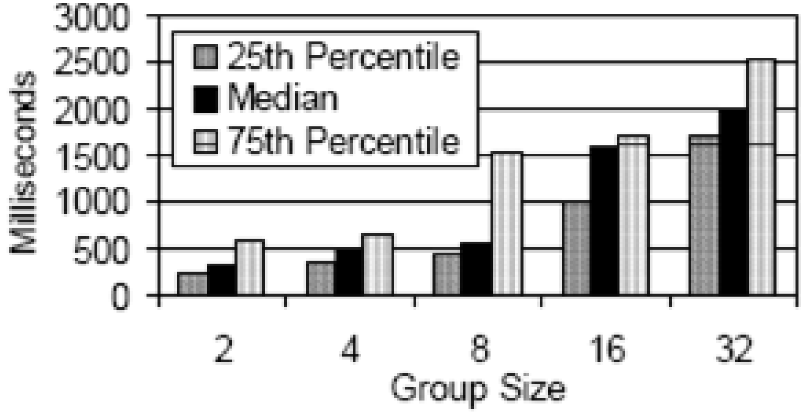


Figure 8. *Latency of Group Creation*

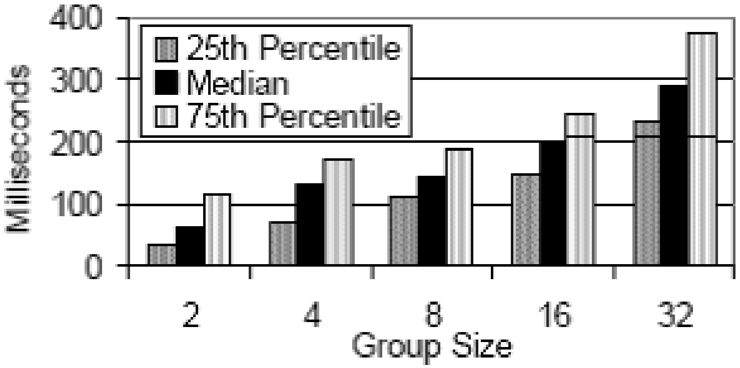


Figure 9. *Latency of Signalled Notification*

Due to open TCP connections
TCP Caches

Experiments

- At steady state, no additional traffic introduced. (However, message size increased by 20 bytes due to hash)
- With churn: with average network size of 300 and an additional 100 nodes churn, FUSE soft notifications result in a 33% increase in messages (Is that good or bad?)
  - Price paid for reusing overlay liveness..
- False positives:
  - Unreliable communication links
    - Under high loss rates more groups failed (obvious)
    - Larger the group size, greater the probability of encountering an unreliable link.
  - Delegate failures: Never generated false positives (due to soft notifications and repair)

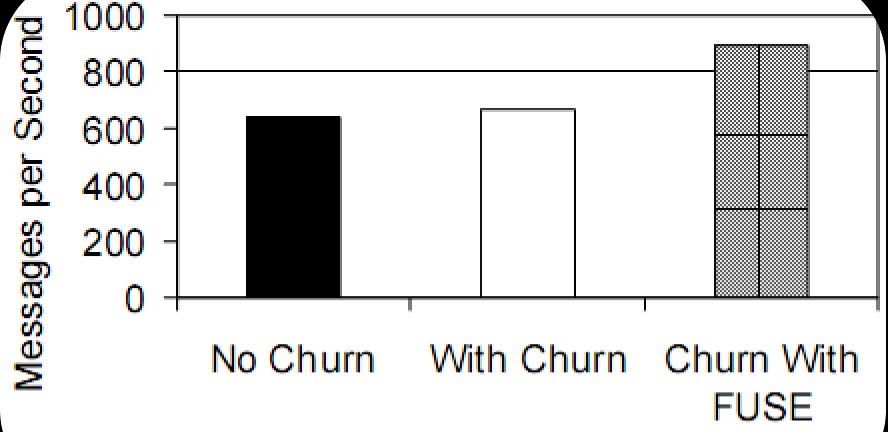


Figure 11. *Costs of Overlay Churn*

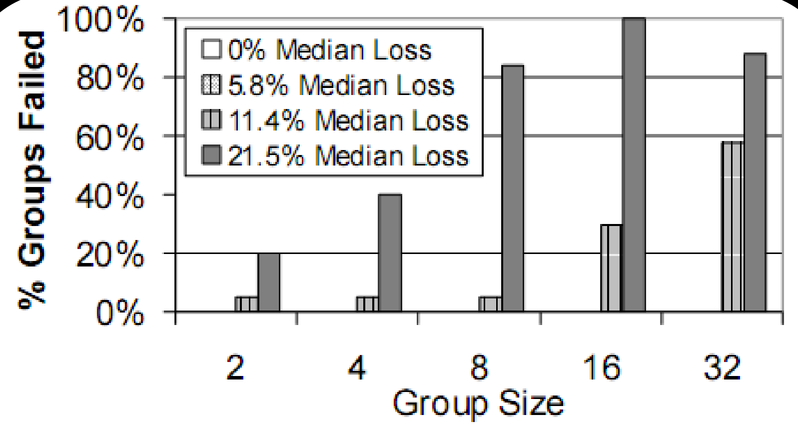


Figure 13. *Group Failures Due to Packet Loss. No failures occurred for 0% and 5.8% loss rates.*

Summary

- Can scale with the number of groups
- Multiple FUSE groups can share liveness checking messages
- Designed to support large number of small and medium sized groups.
- If application already uses a scalable overlay, FUSE can reuse existing liveness checking. Otherwise can implement its own overlay or alternative liveness checking topology.
- Allows applications to declare failures even when application level constraints are violated.
  - FAILURE could mean system failure, violation of application constraints, invalidation of shared data etc. …

Comments

- Is the scalable claim true?
  - Scalable IF implemented on an overlay. Otherwise FUSE does introduce liveness checking traffic.. Implications?
- Cannot be used for consensus.
  - No where did they mention again about solving the consensus problem or did I miss it?
- How to model other failure paradigms like say 'group alive as long as quorum exists'
  - FUSE model always implies that even a single failure implies group failure.
  - Is this kind of implication always suitable?
- Talking about timeouts but did we miss clock synchronization by any chance?