Fault-Tolerant Middleware: Communication

Reliable communication middelware for distributed systems

Kyle E. Benson

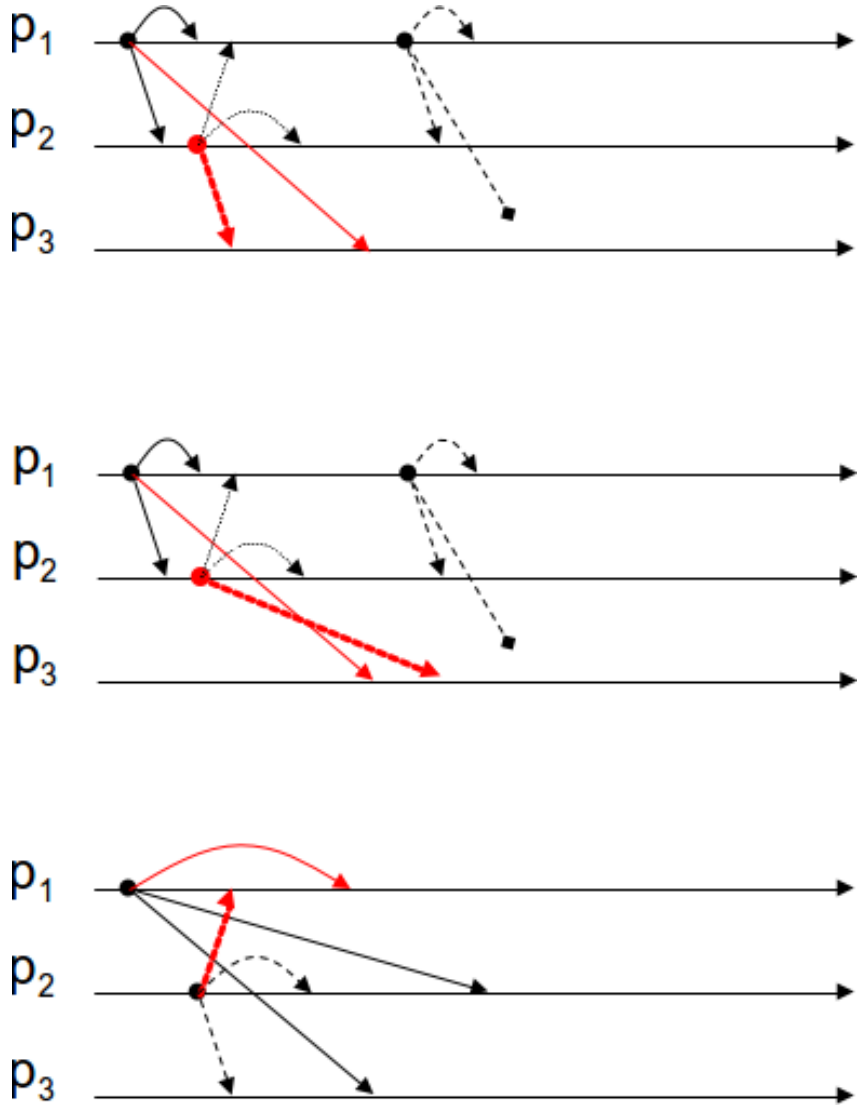
Department of Computer Science
University of California, Irvine
Irvine, California 92697

kebenson@uci.edu

May 30, 2013

Introduction

- Fault-tolerant communication goals
  - **Correctness** of messages, non-corruption guarantee
  - **Ordering** of messages
    - FIFO: If M_a sent before M_b, M_a received before M_b
    - Causal: If M_a causes M_b to be sent, M_a received before M_b at all processes
    - Total: If M_a delivered before M_b at process P_j, M_a delivered before M_b at all other P_i too
  - **Delivery** guarantees, bounds on latency

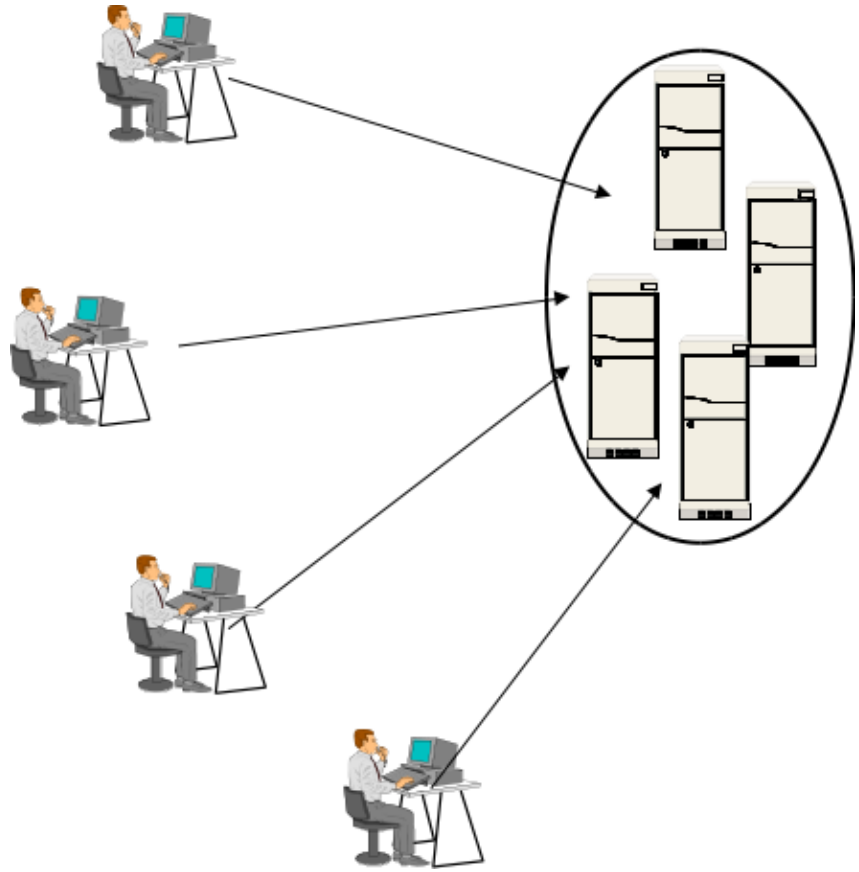


Foundations of Reliability

- How to make unicast reliable?
  - Prevent omission failures
    - Guarantee message delivery
    - Assume correct processes will deliver messages
    - Redeliver on *timeout*
    - No bound on time before reply
  - Guarantee ordering; ignore repeated messages
    - Sequence numbers
    - Timestamps
    - Logical clocks
  - Message integrity
    - Hashing
    - Certificates
    - Keys

Reliable Group Communication

- Why not just use TCP?
  - Consider 100 machines each running 10 processes
  - 1000+ TCP connections at each machine
  - 1 million+ total!
  - Not scalable!
  - Relies on timeouts
  - Ordering harder
  - Similar problems with other client-server connection-oriented protocols
- How to exploit redundancy in communication paths?
- Answer: multicast trees

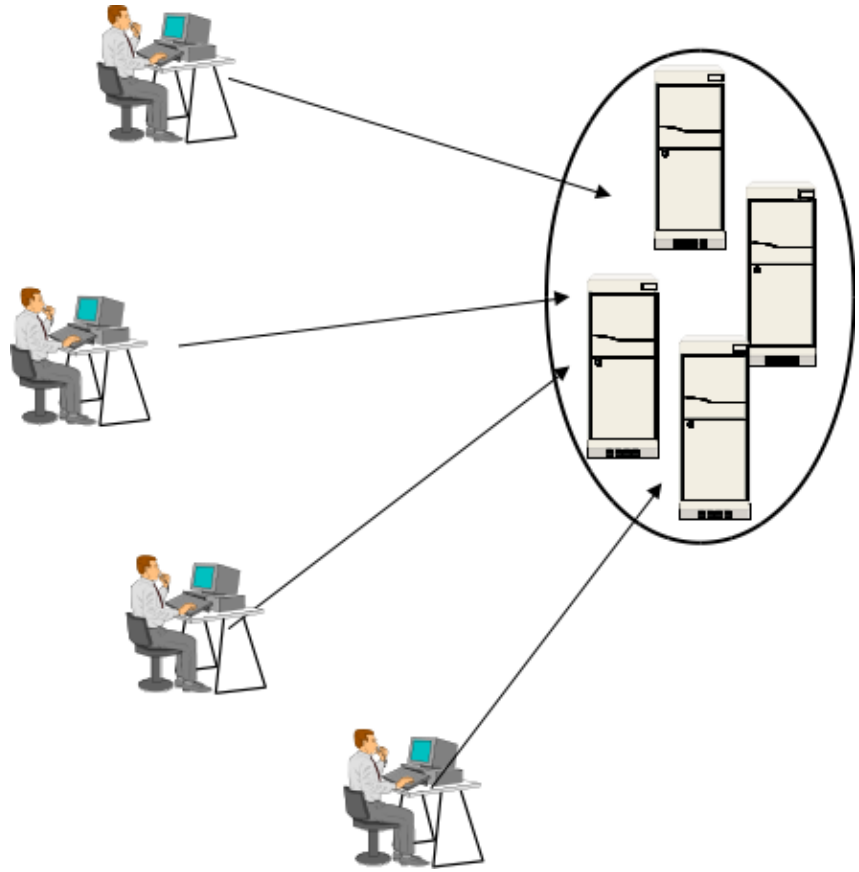


Reliable Distributed Multicast

- **Observation:** distributed systems naturally address groups of processes
  - Coordinating events
  - Replica communication
  - Anycast
  - Reduction
  - Parallel computation
- Distributed process groups $\rightarrow$ multicast groups
  - IP multicast not always supported
  - Make application layer multicast
  - Let the *middleware* handle delivering message to proper groups
  - Decouples machine address from distributed function target
  - More efficient network usage

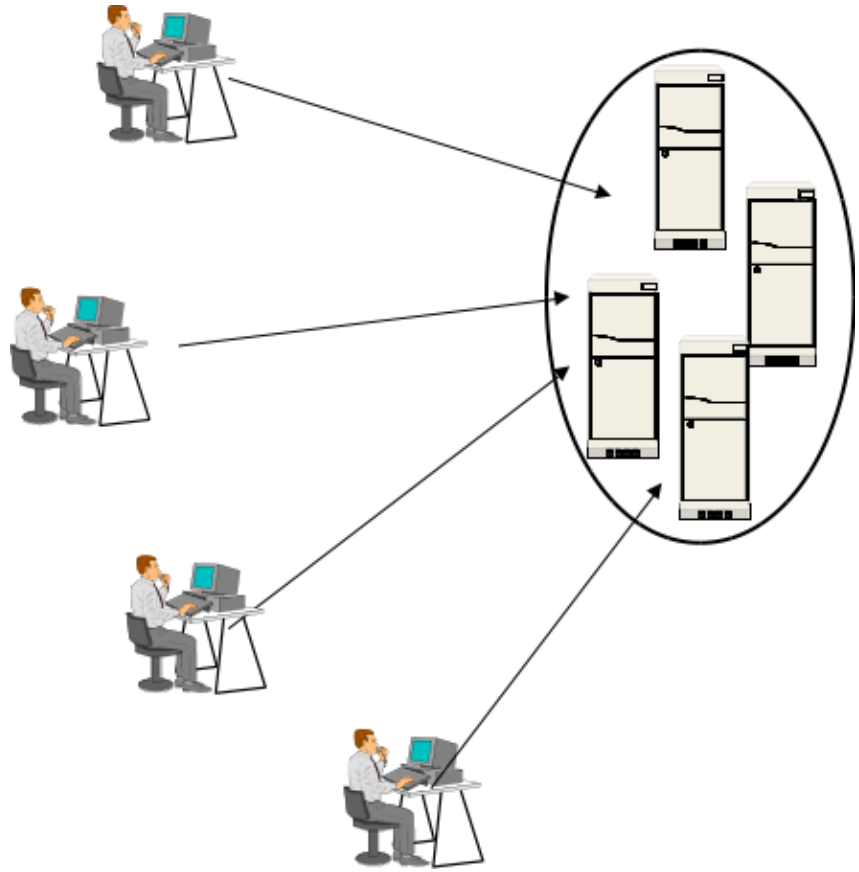
Reliable Group Communication

- Why not just use TCP?
  - Consider 100 machines each running 10 processes
  - 1000+ TCP connections at each machine
  - 1 million+ total!
  - Not scalable!
  - Relies on timeouts
  - Ordering harder
  - Similar problems with other client-server connection-oriented protocols
- How to exploit redundancy in communication paths?
- Answer: multicast trees



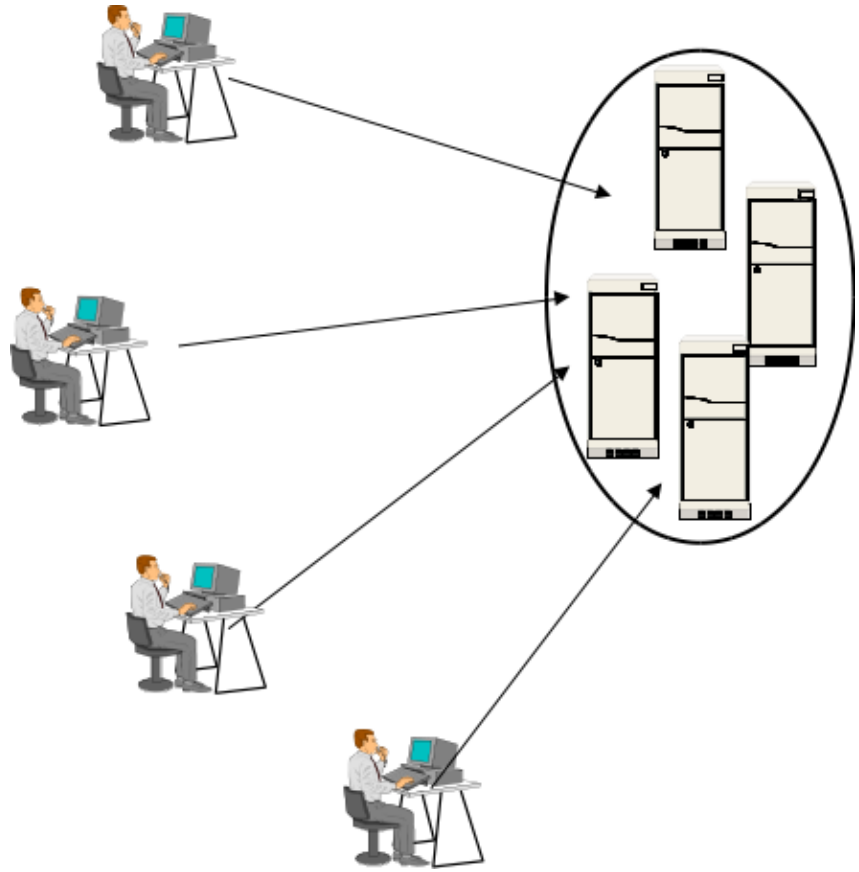
Reliable Group Communication

- Why not just use TCP?
  - Consider 100 machines each running 10 processes
  - 1000+ TCP connections at each machine
  - 1 million+ total!
  - Not scalable!
  - Relies on timeouts
  - Ordering harder
  - Similar problems with other client-server connection-oriented protocols
- How to exploit redundancy in communication paths?
- Answer: multicast trees



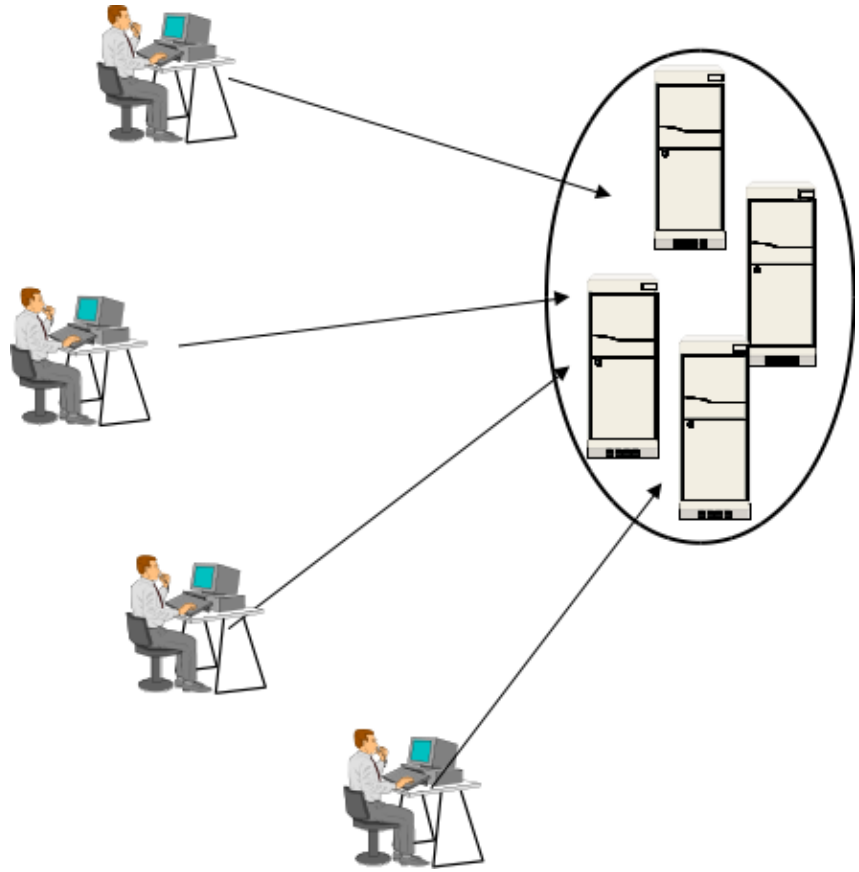
Reliable Group Communication

- Why not just use TCP?
  - Consider 100 machines each running 10 processes
  - 1000+ TCP connections at each machine
  - 1 million+ total!
  - Not scalable!
  - Relies on timeouts
  - Ordering harder
  - Similar problems with other client-server connection-oriented protocols
- How to exploit redundancy in communication paths?
- Answer: multicast trees



Reliable Group Communication

- Why not just use TCP?
  - Consider 100 machines each running 10 processes
  - 1000+ TCP connections at each machine
  - 1 million+ total!
  - Not scalable!
  - Relies on timeouts
  - Ordering harder
  - Similar problems with other client-server connection-oriented protocols
- How to exploit redundancy in communication paths?
- Answer: multicast trees



Reliable Group Communication

- Why not just use TCP?
  - Consider 100 machines each running 10 processes
  - 1000+ TCP connections at each machine
  - 1 million+ total!
  - Not scalable!
  - Relies on timeouts
  - Ordering harder
  - Similar problems with other client-server connection-oriented protocols
- How to exploit redundancy in communication paths?
- Answer: multicast trees