Dynamic and fault-tolerant cluster management

Boris Koldehofe, Anders Gidenstam, Marina Papatriantafilou, and Philippas Tsigas

ÉCOLE POLYTECHNIQUE FÉDÉRALE DE LAUSANNE

CHALMERS
Outline

- Why peer-to-peer resource management is interesting?
  - Large scale event dissemination
  - Ordered event delivery
- Problem description
- Cluster management algorithm
- Properties
- Conclusion and Future Work
Peer-to-peer resource management?

- **Focus**
  - Scalability, reliability, and responsiveness of peer-to-peer services

- **Observe**
  - Many peers may be interested to access similar resources
    - Based on local decision
  - Response time of services depends on the number of peers competing for the service
  - Reliability can only be provided if the number of concurrent peers is limited

- **Approach**
  - To perform an action a process needs to acquire a resource
  - Number of resources are restricted
Example 1: Event dissemination

- Event dissemination / Group communication
  - Scalability and reliability
    - #peers: well addressed by current work
    - #events: ignored
  - Problem: too many events disseminated concurrently
    ⇒ buffer overflow, too many messages per process etc.

- Possible improvement:
  - Restrict number of concurrent senders
  - Number of concurrent peers corresponds to number of peers which are allowed to share a resource in the system
Example 2: Causal event delivery

- Achieved using vector clocks
- Problem: vector clocks grow linearly with the number of peers which send messages
  ⇒ long latencies for large number of processes
- The vector clock is a resource to be used by at most n processes concurrently
- Benefits:
  1. dynamic reuse of vector clock entries
  2. Message sizes stay constant
     ⇒ Scalability
This work

- Resource management for P2P services
  - can improve scalability
  - can improve reliability
- Best applicable where an action of a single peer causes a large number of peers to perform work
- Present a cluster management algorithm
  - Manages resources decentralised
  - Fault-tolerant
Basic Resource Management Model

- Event-based system
  - set of resources $R = \{r_1, \ldots, r_l\}$
  - Using $r_i \sim$ sending event

- Cluster Model:
  - resources are partitioned into several disjoint clusters $C_1, C_2, \ldots$ with $\bigcup_i C_i = R$
  - Cluster manages $n$ distinguishable tickets $t_0, \ldots, t_{n-1}$
  - Process uses a resource only if it obtained a ticket from the cluster managing the resource

- Cluster ensures
  - Never two processes own the same ticket
Cluster Management

- Each cluster corresponds to a process group
- Interested peers join
- Observers – everyone
  - Join the process group
- Using a resource
  - At most \( n \) at a time
  - Core of the cluster
  - \( \sim \) obtain a ticket
Problem description

- Decentralised management of tickets
  - Two processes never own the same ticket
- Fault tolerance
  - Stop failures
  - Communication failures
- Reclaim tickets from failed peers
- Communication paradigm
  - Speed of clocks approximately synchronised
  - Message passing
Cluster Management Algorithm

- Ring Structure
  - peers form a cycle (max n)
  - Predecessor and successor are determined by the ticket a peer obtained
  - Each peer manages entries immediately before it.

- Join
  - Contact any coordinator
  - Notify successor if given an entry
  - Notify all about the new coordinator
Dealing with failures

- Problem: If a process fails need to be able to reclaim vector entries
- Solution idea: Sending alive messages to $2k+1$ successors
- Process to proceed needs to receive $k+1$ alive messages from known processes
- Detect successor failing:
  - Exclusion algorithm contacting the closest successor
  - At the end either initiator succeeds in exclusion or fails
- Can tolerate $k$ failures of $2k+1$ known processes
Basic Idea of Exclusion algorithm

- Two party negotiation not feasible
  - partitioning
- Instead peer determines set of $2k+1$ closest predecessors for its immediate successor
- In each round
  - Send Update($2k+1$ closest predecessors) to immediate neighbours
  - Send ALIVE message to $2k+1$ closest successors

$k=1$

Diagram:
- a -> b -> c -> d -> e -> f
  - UPDATE\{a,b,c\} from c to d
  - UPDATE\{b,c,d\} from e to f
Cont. Exclusion Algorithm

- Determine two sets
  - \( L_p = \{\text{predecessor received by the last UPDATE}\} \)
  - \( R_p = \{\text{predecessors successfully send by last UPDATE}\} \)
  - E.g. \( L_d = \{a, b, c\}, R_d = \{b, c, d\} \)
- Exclusion(p,q) succeeds if
  - \( L_p \cap R_q > k+1 \)
  - \( k+1 \) peers in \( L_p \cap R_q \) confirm exclusion

k=1

\[ \text{UPDATE}\{a,b,c\} \quad \text{UPDATE}\{b,c,d\} \quad \text{Exclude} \]
Algorithms Properties

- Correctness
  - Proof in the paper
- Overhead in messages
  - \(2k+1\) heartbeat messages send in each round
  - Successful ticket acquisition is followed by a Multicast
- Availability of tickets
  - During exclusion of failed tickets coordinators cannot release tickets
- Analysis:
  - \(p_f\): failure rate  \(\alpha\): fraction of taken tickets
  - In equilibrium failing and joining peers:
    - Peer succeeds w.h.p. to acquire a ticket if
    \[p_f < \frac{1}{2} (1-\alpha)\]
Conclusion and Future Work

- Fault-tolerant cluster management model
  - Can support scalable and reliable peer-to-peer services
- Presented an algorithm
  - Decentralised situation
  - Proven correctness in the occurrence failures
    - Stop failures, message omissions
  - Low message overhead
  - Good availability of tickets in the occurrence of failures
- Future work
  - Combining and testing with peer-to-peer services
    - Beyond examples introduced
  - Practical evaluation of algorithms properties
    - Availability of tickets
    - Fairness properties