Autonomous Tool Infrastructure

Dorian Arnold
Department of Computer Science
University of New Mexico

CScADS Petascale Tools Workshop
July 20-24, 2009
The Context:
Tree-based Overlay Networks

MRNet: TBŌN Prototype
Problem Statement: Efficient, Scalable Application Performance

1. How much system-specific knowledge does application (developer) need?

2. How much application-specific knowledge does system (developer) need?

3. How far can we get answering “NONE” and “NONE”?
The Approach: An Autonomous TBŌN Infrastructure

TBŌN Autonomy aka the self-* properties:

• **Self-configuring**
  - Automatic TBŌN topology configuration

• **Self-monitoring**
  - TBŌN health and performance

• **Self-healing**
  - TBŌN fault tolerance and failure recovery

• **Self-optimizing**
  - Dynamic TBŌN reconfiguration to improve performance

Must maintain scalable, efficient performance!
Research Challenges

• How can we provide a reliable TBŌN service in the presence of failures?

• How do we choose the “best” TBŌN topologies?
  – Application load and system characteristics may vary over time

• How can we dynamically improve TBŌN performance?
  – Throughput, latency, resource consumption, startup costs, …

• Can we design a flexible, elegant solution space?
Outline: Past, Present and Future Directions

• Past:
  – TBŌN event/failure detection
  – TBŌN failure recovery

• Present and future
  – TBŌN performance monitoring
  – TBŌN performance modeling
  – Dynamic TBŌN self-configuration and optimization
  – Other issues (as time permits)
Recent MRNet Developments

• Before:
  – MRNet only supported static topologies
  – MRNet did not tolerate any failures

• As of MRNet 2.0 (August ’08)
  – Event detection service
    • Failure detection
    • Dynamic topology configuration
    • New MRNet instantiation protocol
  – State composition for failure recovery
Event Detection Service (EDS) thread
  - In each MRNet process

  - Passive detection of asynchronous events
    • Failure events for failure detection
    • Connection events for dynamic reconfiguration

  - Connection-based (TCP) mechanisms
    • Monitor watch list of event sockets
      - Listening socket
      - New Failure Detection Connection protocol message
      - New Data Connection protocol message
Self-monitoring: Detecting Functional Failures

- Each process monitors its peers (parent and children)
- Connect to peer EDS
- **Send** New Failure Detection Connection message
- Add failure detection event sockets to watch list
- Socket error → peer failure
1. The MRNet tree must be reconfigured to reconnect orphaned subtrees

2. MRNet must recover any lost process or channel state (that it can)
At initialization or after failures, orphan connect to new parent’s EDS

- **Send** New Data Connection **protocol** message

- Child/parent establish data socket
State Compensation

– Compensate for lost state using inherently redundant information from survivors

– Avoid overhead of explicit data replication

– State Composition
  • Lightweight mechanism
  • Requires associativity, commutativity and idempotence
What’s Next?
“Performance Failures”

• What is a performance failure?
  – Generally, employing a sub-optimal topology

  – Realizing (much) less than optimal performance
    • Data aggregation latency and throughput
    • Resource under-utilization
    • Imbalanced topologies

  – Per application?
  – Per flow/stream?
“best” topology dependent upon
  – Participating end-points
  – Data aggregation operation
  – Application data rate
  – …

“best” is different for different streams!
  – How can we efficiently enable different topologies for different flows
TBŌN Components for Autonomy

Sensors: hw/sw characteristics, runtime events, etc.
Monitoring: collecting/correlating events to identify patterns and symptoms, e.g., threshold checking.
Detecting: evaluate symptoms to determine if problems exist and if action is necessary, e.g., do we have a bottleneck?
Deciding: determine the best course of action to modify the topology.
Acting: effect the recommended topology changes.

Key Challenges:
- Decentralization
- Low (background) overhead
- Rapid execution
- Must provide more benefits than drawbacks!
Other Issues: Many MRNets or 1

<table>
<thead>
<tr>
<th></th>
<th>1 Network per Application</th>
<th>1 Network shared across applications</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Pros</strong></td>
<td>Simple</td>
<td>Fast startup</td>
</tr>
<tr>
<td></td>
<td>Ease-of-deployment</td>
<td>Better resource utilization</td>
</tr>
<tr>
<td></td>
<td>No interference</td>
<td></td>
</tr>
<tr>
<td></td>
<td>between applications</td>
<td></td>
</tr>
<tr>
<td><strong>Cons</strong></td>
<td>Slow startup</td>
<td>More complex</td>
</tr>
<tr>
<td></td>
<td>Poorer resource utilization</td>
<td>Persistent network</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Help address collocation problems</td>
</tr>
</tbody>
</table>
Other Issues: Native Services

• Separate service dependent from service independent mechanisms

• Improved portability and performance

• Process launching
  – Currently rsh-based
  – Leverage native resource manager or job launcher
    • Might we gain enough startup performance improvement to forego persistent, shared network?

• IPC
  – Currently TCP-based
  – Leverage faster, local communication services
What this means to you

• Simpler, yet better, TBŌN infrastructure
  – Doing (much) more with less!

• We built it, you should come.

Questions?
- Scalable data multicast and aggregation
- Flexible topologies
- User-defined filters
- Trade-off: extra processing nodes for performance
Use tree structure for efficient global dissemination

- Failure report:
  - 32 bits: \{failed rank\}
- Reconfiguration report:
  - 64 bits: \{child rank, parent rank\}
  - Disconnected subtrees intact

- Disseminating process sends to parent and children

- Receiving processes send to peers other than source
outPacket get_FilterState( void ** inFilterState );

- Inputs pointer reference to stream’s filter state
- Outputs “packetized” version of filter state

int load_FilterState( const char * inSharedObject
                       const char * inFilterFunction );

- Used to dynamically load new filter functions
- Also queries for get_FilterState routine
  - If found, filter is recoverable