A Sleep-based Communication Mechanism to Save Processor Utilization in Distributed Streaming Systems

Shoaib Akram Angelos Bilas

Foundation for Research and Technology - Hellas (FORTH)
Institute of Computer Science (ICS)

May 1, 2011

Efficiency in Back-end Processing

- Efficiency in back-end processing is important.
- Scalability is important but software stacks of indiviual nodes are becoming complex :
  - Runtime bloat (Nick Mitchell).
  - Complex messaging protocols.
  - Layers of software, libraries etc.
- This leads to over-provisioning of resources for back-end processing.

Distributed Streaming Systems

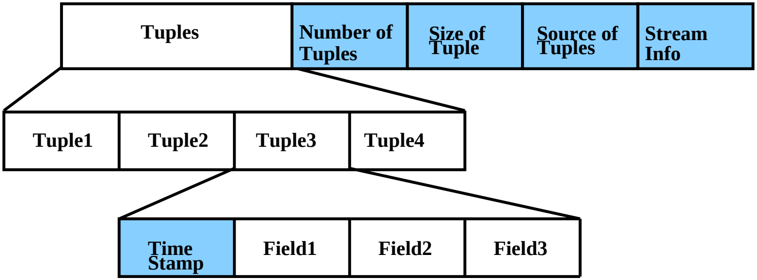
- Recently gaining attention due to large amounts of data to be processed/filtered.
- Static queries and moving data.
- Similar operators like traditional data bases.
- Reasons for adopting a distributed model :
  - Geographically distributed sources of data.
  - Speed-up of application queries.
- Borealis (academic consortium) and SystemS (IBM) are common examples.

Key Requirements of Distributed Streaming Systems

- Scalability to many nodes.
- Provisioning for heavy inter-node communication.
- Rich library of stream operators.
- Communication protocol and operators should be decoupled.

The Architecture of Borealis - Event Structure

- Event-driven architetcure.
- The notion of streams and tuples.



The Architecture of Borealis - Threads and Data Structures

- Four threads that work asynchronously:
  - receive thread
  - process thread
  - prepare thread
  - send thread
- Data structures for inter-thread communication.

Communication Subsystems in Distributed Middleware Systems

- Send/Receive operations are implemented using:
  - Interrupts - High overhead at high network speeds and large message rates.
  - Polling - Wastes CPU cycles at low network rates.
- Send/Receive API provided by Linux Sockets :
  - Blocking sockets (interrupts).
  - Non-blocking sockets (polling).
  - Monitoring multiple sockets (blocking call to select).
- Problems with monitoring multiple sockets with select.

Sleeping - An Alternative Approach

- Sleep for a specific amount of time if no communication is expected.
- Regulation of sleeping time :
  - Kernel issues.
  - Multiple applications.
  - Parameters of a single application changes.
  - Granularity of sleeping time may change with a different kernel.

Our Approach: Distribution/Accumulation of Work

- Typical configuration of a data streaming system is a pipeline of senders/receivers.
- Send and receive threads work asynchronously.
- Goal of send thread :
  - Node downstream has enough work to perform.
- Goal of receive thread :
  - Unpack the events and give work to process thread.
  - Layers above the communication protocol have enough work to do.

Working in Waves

- Both send and receive threads maintain messaging queues.
- The receive thread informs the send thread of the availability of free slots in the queue by sending a message (credit message).
- After processing a few buffers, the receive thread sends a credit message to the send thread.
- The credit message allows the send thread to send data in buffers that the receive thread has already made available.
- If there the send thread can not find a credit message, it sleeps.

Working in Waves

- The receive thread unpacks the events, hand the events to the event handler and then checks for an event in the next slot in the queue.
- If the receive thread can not find data in the buffer, it sleeps.
- While it is sleeping, the send thread fills up the queue with new events.

Working in Waves: Summary

- Sleeping criteria for send thread :
  - Criteria: Sleep for a fixed amount of time if no credits available.
  - Rationale: Receiver is busy unpacking messages and will send credits at some point.
- Sleeping criteria for receive thread :
  - Criteria: Sleep for a fixed amount of time if no new message is available.
  - Rationale:
    - All the available messages were unpacked and distributed to layer above.
    - Processing is much heavier than unpacking.
    - Collect work while consuming no extra CPU cycles.

Machine Parameters and Benchmark for Evaluation

- Four server-type systems running Linux CentOS release 5.4.
- Two Intel Xeon Quad-core (2-way hyper threaded).
- 14 Gbytes DRAM.
- 10 Gbits/s Ethernet NIC from Myrinet.
- 10 Gbits/s Ethernet HP ProCurve 3400cl switch.
- A custom-benchmark that filters the incoming data (filter condition is always true to load network).
- First node generates the tuples, the next two process the tuples.
- The last node receives the tuples and consumes them internally.

Some Parameters of Borealis

- No. of instances of borealis (8).
- Batching factor (varying).
- Tuple size (varying).
- Size of send-side queue (10).
- Size of receive-side queue (100).
- Frequency of exchanging credits (every 10 buffers).
- Sleeping time is 10 ms.

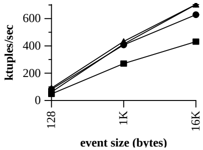
Myrinet MX - A User-level Networking API

- Provides a user-level networking API.
- Baseline throughput is higher :
  - Removes one copy on send side.
  - Removes two copies on the receive path.
  - Reduces the number of interrupts on the receive side.
- Fine-grained control for managing buffers.
- Ease of implementation of flow-control mechanisms.

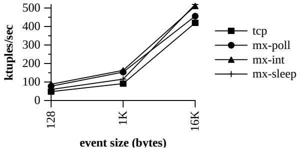
Our Configurations of Borealis for Evaluation

- tcp : Baseline version of borealis with TCP/IP.
- mx-poll : Borealis with Myrinet MX protocol and polling operations for testing buffers.
- mx-int : Borealis with Myrinet MX protocol and polling opeations for testing buffers.
- mx-sleep : Borealis with Myrinet MX protocol and using sleep system call.

Baseline Throughput of Borealis with TCP and MyrinetMX



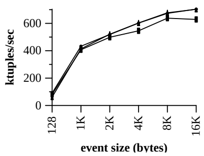
(a) 128 bytes



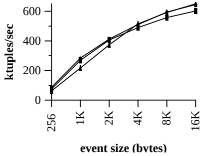
(b) 512 bytes

- mx-int improves throughput of borealis compared to tcp (22%).
- mx-poll has lower throughput compared to mx-int.
- mx-adp gives better throughput compared to tcp (23-63%).

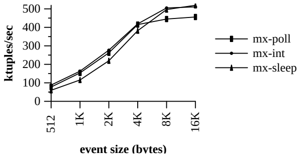
Throughput and CPU Utilization - All Configurations



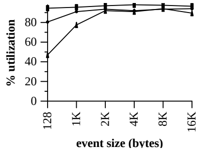
(c) 128 bytes



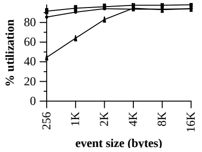
(d) 256 bytes



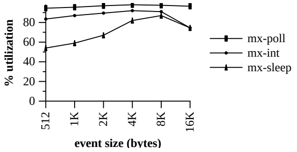
(e) 512 bytes



(f) 128 bytes



(g) 256 bytes



(h) 512 bytes

General Trends in Writing Middleware Systems

- Modules are written by different developers.
- Accounting for heterogenous architectures.
- Accounting for slow networks.
- Over-provisioning for memory.

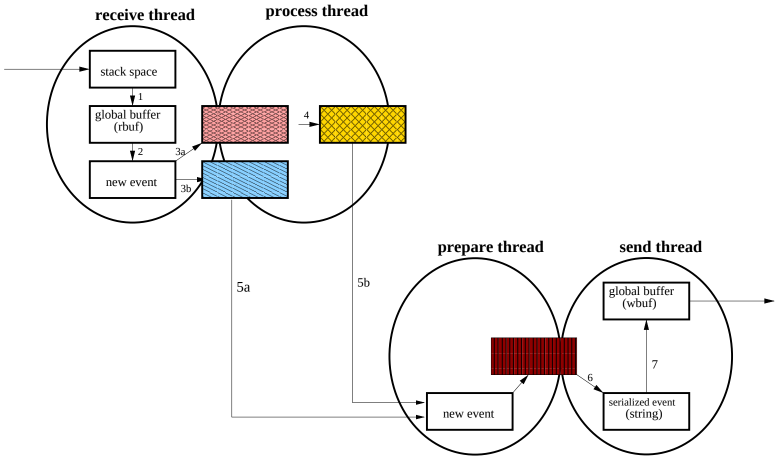
General Trends in Writing Middleware Systems

- Buffer management across threads/modules :
  - (buffer_ptr,size).
  - Copying a buffer and passing it.
- Serialization-Deserialization - Heterogenity and Portability :
  - Communication among heterogenous nodes.
  - Packing data-structures spread in different parts of memory.
  - Overhead of copies.
  - Use separate send operation to send each field of data-structure.

General Trends in Writing Middleware Systems

- Message Queuing for Asynchronous Operation :
  - Threads might block on slow networks.
  - Buffering provides asynchronous operation.
  - Not necessary on fast networks.
  - Send the event from the prepare thread and block (in case).
- Flow Control :
  - Memory is usually over-provisioned.
  - Virtual memory is backed up by swap space on disk.
  - Proper flow-control involves accounting memory under utilization (by different threads).
  - Proper inter-thread flow-control saves memory resources for other tasks in the system.
  - Different structures could possibly allow flow-control (which one to choose).

Observations from the Borealis Communication Flow



Conclusions

- Sleep-based communication policies can save CPU cycles for other tasks.
- Main problem is to find a criteria to sleep.
- Portability is a concern.
- Save CPU cycles for a given application :
    - Less power.
- Give CPU cycles to some other application :
    - Improves (overall) energy efficiency of a system.
- Too much focus on scaling?

Conclusions

- Sleep-based communication policies can save CPU cycles for other tasks.
- Main problem is to find a criteria to sleep.
- Portability is a concern.
- Save CPU cycles for a given application :
  - Less power.
- Give CPU cycles to some other application :
  - Improves (overall) energy efficiency of a system.
- Too much focus on scaling?