A Distributed Shared Buffer Space for Data-intensive Applications

Renaud Lachaize
Sardes Project
Inria Rhône-Alpes
Grenoble, France

Jørgen Sværke Hansen
Department of Computer Science
University of Copenhagen, Denmark

5th Distributed Shared Memory Workshop (DSM 2005) – Cardiff, UK - 2005-05-11

Context

- Distributed applications/services running on clusters of servers
- Involved with intensive data transfers
- Work developed for a cooperative storage framework
  - We speculate that it may also be useful for other target applications (e.g. multimedia servers)

Motivation (1/2)

- Data servers are hard to tune and optimize in a generic fashion
  - Several parameters may vary significantly:
    - Node & network configurations
    - Topology: central server, symmetric, …
      - Some nodes may act only as "message routers" that do not access the data
    - Traffic patterns
      - Workload characteristics
      - Maximize client performance (throughput / latency)
      - Minimize load on server nodes
      - Constraints for control messages and data transfer may differ

  => Need for several data transfer strategies

Motivation (2/2)

- Maintaining multiple versions of the server's code is complex
  - Data transfers impact most parts of the code
- Using a hardware-independent communication API is not enough
  - Either no flexibility regarding the transfer strategy for a given interconnect family
  - Or only considering "direct" communication between two hosts
    - What about router nodes ?

Key Ideas

- "True" Distributed Shared Memory (DSM) systems have usually been employed to program HPC applications easily
- We propose a restricted form of DSM for flexible data exchange among servers
  - Coherency issues are left to the application
- Main principle : decoupling control messages and data transfers

Outline

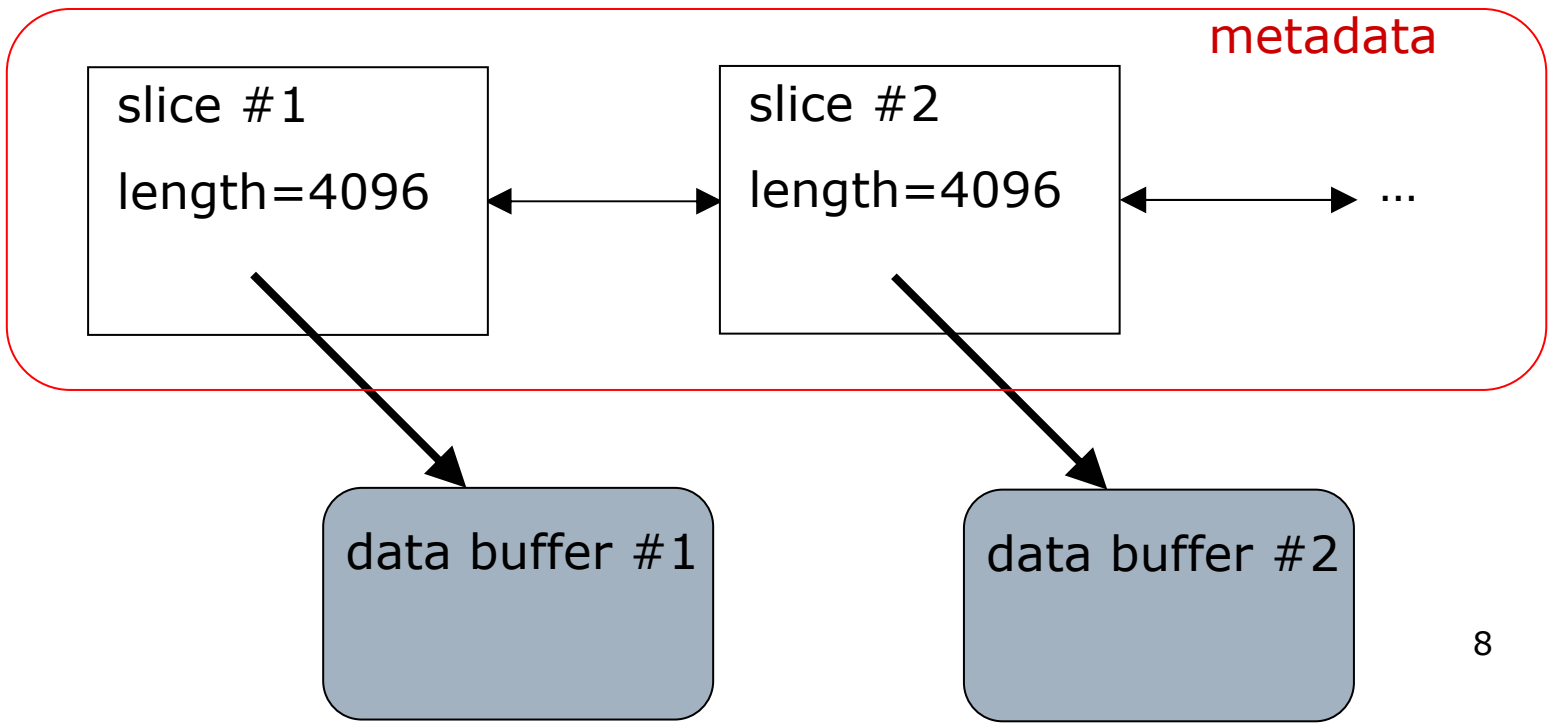
- Introduction
- Concepts
  - Buffer Aggregates
  - Transfer modules
  - Putting it all together
- Programming model
- Examples & experiments

Application data exchange

- Components of a distributed application exchange data buffers through a shared buffer space
- Buffers are explicitly exported to and imported from this shared buffer space at the application level
- Shared buffers are represented by **buffer aggregates**

Buffer Aggregate (BA)

- Ordered list of data buffers
- Helps manipulation of data without memory copy operations (appending headers, combining/splitting buffers, …)



Buffer aggregate (2)

- Concept and API close to previous work such as IOLite
  - create empty BA, import buffers from the OS, append a slice to the BA, get context object, get next slice from context object, …
  - Unlike IOLite, current prototype does not provide completely unified buffering and caching within an OS (only within the framework)

Global buffer address space

- A buffer aggregate is associated to a home node (who created it)
- The contents of the BA can be mapped to a cluster wide address space
- A copy of the global metadata can be communicated to other nodes

- On a given node, at a given time, we can have:
  - a global BA instance (using global data buffer addresses)
    → reference to data slices on the home node

    and/or
  - a local BA instance (using local pointers to data buffers)
    → enable local access to the slices

- All updates to the buffer contents are eventually propagated to the home node

Buffer mappings

- Establishment/removal of a local mapping are managed by the framework
- Programmer indicates need for local access to the data buffers through request for a context object
- BA framework relies on underlying code module (IODSM) for data transfers

Different IODSM implementations can coexist

- Every implementation is associated with both
  - a communication API
    - hardware dependent (eg. Myrinet GM, SCI IRM) or not (eg. sockets)
  - a transfer strategy, defined by:
    - a transfer mechanism
      - "real data copy": data packet, RDMA
      - remote memory mapping (SCI)
    - a communication pattern (push/pull)

IODSM implementations

- Any IODSM must implement the following interface:
  - map_local_to_iodsm
  - map_iodsm_to_local
  - update_local_from_iodsm
  - update_iodsm_from_local
  - unmap_local
  - unmap_global
- These functions are only called by the BA infrastructure, not by the application programmer

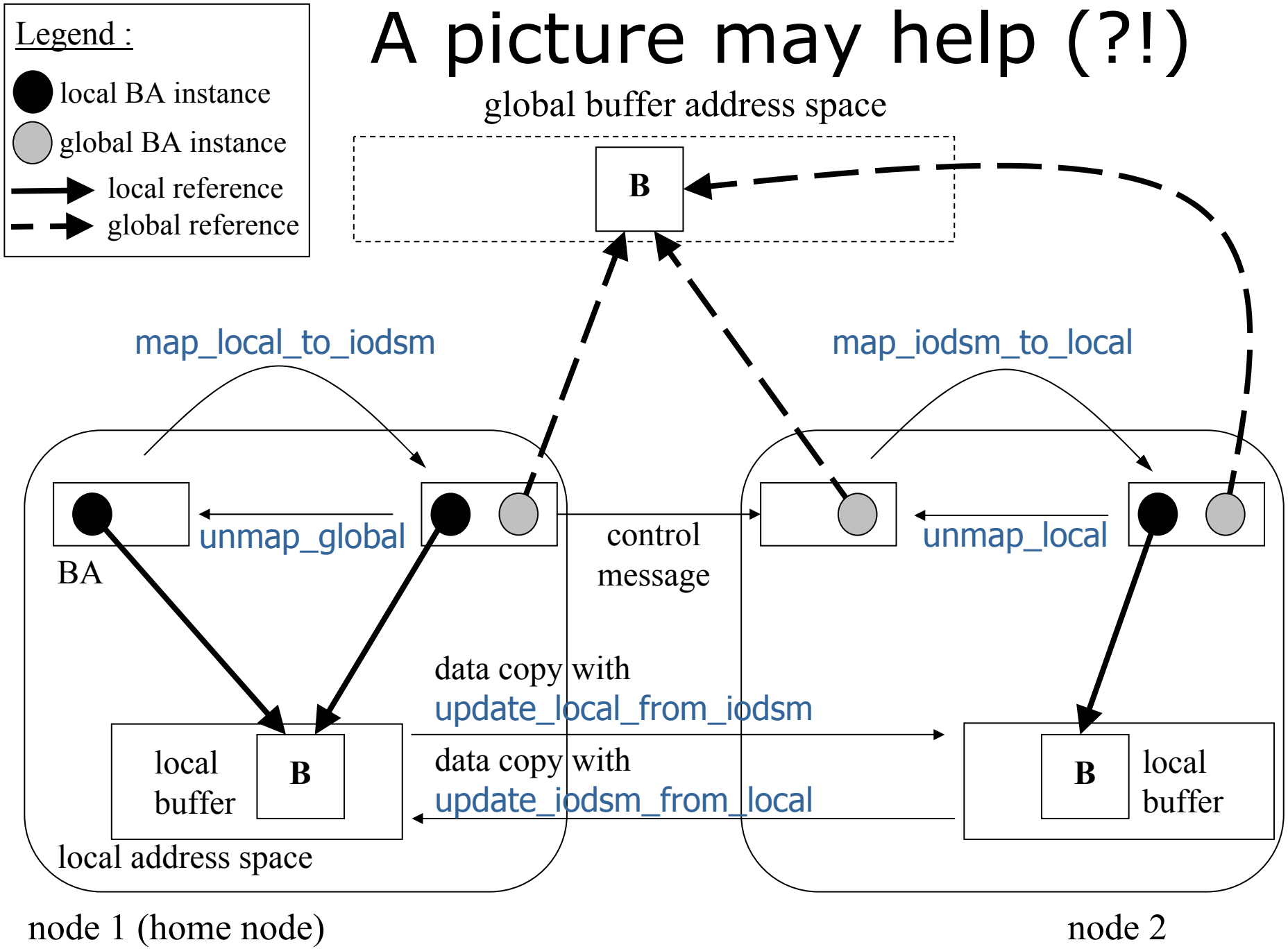
Putting it all together

- At the application level, nodes exchange control messages
  - Network independence achieved
    - with a hardware-independent API
    - or with a modular structure for the application
  - A control message includes one (or several) global buffer aggregate(s)

- BA serialization
  - Default mechanism only serializes global BA instance
  - Other mechanisms can be defined

    => Decoupling between control messages and data transfers is not mandatory

A picture may help (?!)



Outline

- Introduction
- Concepts
  - Buffer Aggregates
  - Transfer modules
  - Putting it all together
- Programming model
- Examples & experiments

Programming model (1/3)

- **Node A (client)**

  prologue

  b = iobuf_import(buffers)

  iobuf_globalize(b)

  p = iobuf_pack(b)

  send(data_request + p)

Programming model (2/3)

- **Node B (server)**

  receive(data_request + p)

  d = iobuf_unpack(p)

  c = iobuf_gen_start(d, local)

  perform data operations

  iobuf_gen_end(c)

  acknowledge request

  iobuf_destroy(d)

Programming model (3/3)

- **Node A (client)**

  receive ack

  iobuf_unglobalize(b)

  epilogue

  iobuf_destroy(b)

Examples and experiments

- 2 Examples on Gigabit Ethernet illustrating the benefits of decoupling control and data flow
  - Assuming direct communication is possible between any pair of nodes
- Overhead of the framework on SCI and Gigabit Ethernet
- Test setup:
  - Athlon 1800, 1 GB DDRAM, AMD 760MP chipset, Broadcom 5701 GigE and Dolphin D330 adapters, linux kernel 2.4.20

Example 1: Delayed allocation of data payload

- Server managing different priorities for data-sending clients
  - Low priority requests can accumulate at the server
  - A pull-based write approach can help lower the memory footprint on the server

- Experiment with 1 high priority and 3 other clients (all write-intensive)
  - Delayed allocation helps reducing the memory load on the server
    - From 778 to 335 MB (56% lower)
  - Moderate increase in latency (9%) compared to joint request and data transfers

Example 2: Elimination of data copies on router nodes

- Central server routing client (read & write) requests to back ends
  - Decoupling control and data flow allows to bypass the router for data transfers
    - Latency reduction up to:
      - 9% for small transfers (4-8 kB)
      - 43% for bulk transfers (64-128 kB)

Overhead of the framework

- Reengineering of our distributed storage application
  - For a given transfer strategy, performance comparison of IODSM based version versus a "hard-wired" version
    - Gigabit-Ethernet : packet-based, sender-driven strategy
    - SCI : RDMA, server-driven strategy
- Various I/O benchmarks with cold caches
  - mkfs, file and directory copy, Bonnie, linux kernel compilation, etc.
  - monitoring of latency of individual I/O requests and total running time
- Results
  - 3-5% overhead, for small (4-8 kB), sequentially synchronized requests
  - No noticeable overhead otherwise

Conclusion

- Proposition : a programming model and the associated framework that allow
  - to tune the data transfers for a given setup (networking hardware, topology)
  - by (possibly) decoupling control messages and data transfers
  - with no modification to the core applicative code and an acceptable overhead
- Perspectives
  - Dynamically choose transfer strategy according to the operating conditions
    - Handled either with a meta-IODSM or at the application level
  - Open issue : support for a wider range of data access patterns (e.g. partial updates of buffers)