# DDSS: A Low-Overhead Distributed Data Sharing Substrate for Cluster-Based Data-Centers over Modern Interconnects

Karthikeyan Vaidyanathan, Sundeep Narravula, and Dhabaleswar K. Panda

Department of Computer Science and Engineering
The Ohio State University
{vaidyana, narravul, panda }@cse.ohio-state.edu

**Abstract.** Information-sharing is a key aspect of distributed applications such as database servers and web servers. Information-sharing also assists services such as caching, reconfiguration, etc. In the past, information-sharing has been implemented using ad-hoc messaging protocols which often incur high overheads and are not very scalable. This paper presents a new design for a scalable and a low-overhead *Distributed Data Sharing Substrate* (DDSS). DDSS is designed to support efficient data management and coherence models by leveraging the features of modern interconnects. It is implemented over the OpenFabrics interface and portable across multiple interconnects including iWARP-capable networks in LAN/WAN environments. Experimental evaluations with networks like InfiniBand and iWARP-capable Ammasso through data-center services show an order of magnitude performance improvement and the load resilient nature of the substrate. Application-level evaluations with Distributed STORM achieves close to 19% performance improvement over traditional implementation, while evaluations with check-pointing application suggest that DDSS is highly scalable.

## 1 Introduction

Distributed applications in the fields of nuclear research, biomedical informatics, satellite weather image analysis etc., are increasingly getting deployed in cluster environments due to their high computing demands. Advances in technology have facilitated storing and sharing of the large datasets that these applications generate, typically through a web interface forming web data-centers [1]. A web data-center environment (Figure 1) comprises of multiple tiers; the first tier consists of front-end servers such as the proxy servers that provide services like web messaging, caching, load balancing, etc. to clients; the middle tier comprises of application servers that handle transaction processing and implement business logic, while the back-end tier consists of database servers that hold a persistent state of the databases and other data repositories. In order to efficiently host these distributed applications, current data-centers also need scalable support for intelligent services like dynamic caching of documents, resource management, load-balancing, etc. Apart from communication and synchronization, these applications and services exchange some key information at multiple sites (e.g, timestamps of cached copies, coherency and consistency information, current system load). However, for the sake of availability, high-performance and low-latency, programmers use ad-hoc messaging protocols for maintaining this shared information. Unfortunately, as

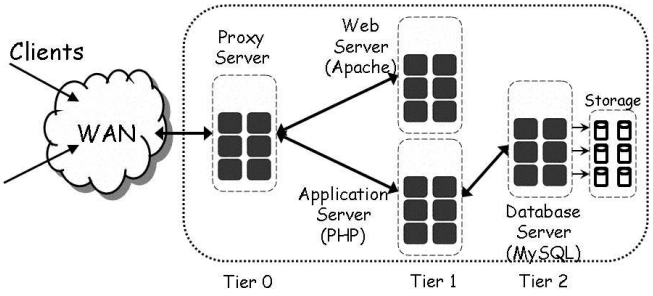


**Fig. 1.** Web data-centers

mentioned in [2], the code devoted to these protocols accounts for a significant fraction of overall application size and complexity. As system sizes increase, this fraction is likely to increase and cause significant overheads.

On the other hand, System Area Network (SAN) technology has been making rapid progress during the recent years. SAN interconnects such as InfiniBand (IBA) [3] and 10-Gigabit Ethernet (10GigE) have been introduced and are currently gaining momentum for designing high-end computing systems and data-centers. Besides high performance, these modern interconnects provide a range of novel features and their support in hardware, e.g., Remote Direct Memory Access (RDMA), Atomic Operations, Offloaded Protocol support and several others. Recently OpenFabrics [4] has been proposed as the standard interface that allows protable implementations over several modern interconnects like IBA, and iWARP capable ethernet interconnects including [5] Chelsio, Ammasso [6], etc., both in LAN/WAN environments.

In this paper, we design and develop a low-overhead distributed data sharing substrate (DDSS) that allows efficient sharing of data among independently deployed servers in data-centers by leveraging the features of the SAN interconnects. DDSS is designed to support efficient data management and coherence models by leveraging the features like one-sided communication and atomic operations. Specifically, DDSS offers several coherency models ranging from null coherency to strict coherency.

Experimental evaluations with IBA and iWARP-capable Ammasso networks through micro-benchmarks and data-center services such as reconfiguration and active caching not only show an order of magnitude performance improvement over traditional implementations but also show the load resilient nature of the substrate. Application-level evaluations with Distributed STORM using DataCutter achieves close to 19% performance improvement over traditional implementation, while evaluations with checkpointing application suggest that DDSS is scalable and has a low overhead. The proposed substrate is implemented over the OpenFabrics standard interface and hence is portable across multiple modern interconnects.

## 2 Constraints of Data-Center Applications

Existing data-center applications such as Apache, MySQL, etc., implement their own data management mechanisms for state sharing and synchronization. Databases communicate and synchronize frequently with other database servers to satisfy the coherency and consistency requirements of the data being managed. Web servers implement complex load-balancing mechanisms based on current system load, request

patterns, etc. To provide fault-tolerance, check-pointing applications save the program state at regular intervals for reaching a consistent state. Many of these mechanisms are performed at multiple sites in a cooperative fashion. Since communication and synchronization are an inherent part of these applications, support for basic operations to read, write and synchronize are critical requirements from the DDSS. Further, as the nodes in a data-center environment experience fluctuating CPU load conditions the DDSS needs to be resilient and robust to changing system loads.

Higher-level data-center services are intelligent services that are critical for the efficient functioning of data-centers. Such services require sharing of some state information. For example, caching services such as active caching [7] and cooperative caching [8], [9] require the need for maintaining versions of cached copies of data and locking mechanisms for supporting cache coherency and consistency. Active resource adaptation service requires the need for advanced locking mechanism in order to reconfigure nodes serving one website to another in a transparent manner and needs simple mechanism for data sharing. Resource monitoring services, on the other hand, require efficient, low overhead access to the load information on the nodes. The DDSS has to be designed in a manner that meets all of the above requirements.

## 3 Design Goals of DDSS

To effectively manage information-sharing in a data-center environment, the DDSS must understand in totality, the properties and the needs of data-center applications and services and must cater to these in an efficient manner.

Caching dynamic content at various tiers of a multi-tier data-center is a well known method to reduce the computation and communication overheads. Since the cached data is stored at multiple sites, there is a need to maintain cache coherency and consistency. Broadly, to accommodate the diverse coherency requirements of data-center applications and services, DDSS supports a range of coherency models. The six basic coherency models [10] to be supported are: 1) *S*trict Coherence, which obtains the most recent version and excludes concurrent writes and reads. Database transactions require strict coherence to support atomicity. 2) *W*rite Coherence, which obtains the most recent version and excludes concurrent writes. Resource monitoring services [11] need such a coherence model so that the server can update the system load and other load-balancers can read this information concurrently. 3) *R*ead Coherence is similar to write coherence except that it excludes concurrent readers. Services such as reconfiguration [14] are usually performed at many nodes and such services dynamically move applications to serve other websites to maximize the resource utilization. Though all nodes perform the same function, such services can benefit from a read coherence model to avoid two nodes looking at the same system information and performing a reconfiguration. 4) *N*ull Coherence, which accepts the current cached version. Proxy servers that perform caching on data that does not change in time usually require such a coherence model. 5) *D*elta coherence guarantees that the data is no more than $x$ versions stale. This model is particularly useful if a writer has currently locked the shared segment and there are several readers waiting to the read the shared segment. 6) *T*emporal Coherence guarantees that the data is no more than $t$ time units stale.

Secondly, to meet the consistency needs of applications, DDSS should support versioning of cached data and ensure that requests from multiple sites view the data in a

consistent manner. Thirdly, services such as resource monitoring require the state information to be maintained locally since the data is updated frequently. On the other hand, services such as caching and resource adaptation can be cpu-intensive and hence require the data to be maintained at remote nodes distributed over the cluster. Hence, DDSS should support both local and remote allocation in the shared state. Due to the presence of multiple threads on each of these applications at each node in the data-center environment, DDSS should support the access, update and deletion of the shared data for all threads in a transparent manner. DDSS should also provide asynchronous interfaces for reading and writing the shared information Further, as mentioned in Section 2, DDSS must be designed to be robust and resilient to load imbalances and should have minimal overheads and provide high performance access to data. Finally, DDSS must provide an interface that clearly defines the mechanism to allocate, read, write and synchronize the data being managed.

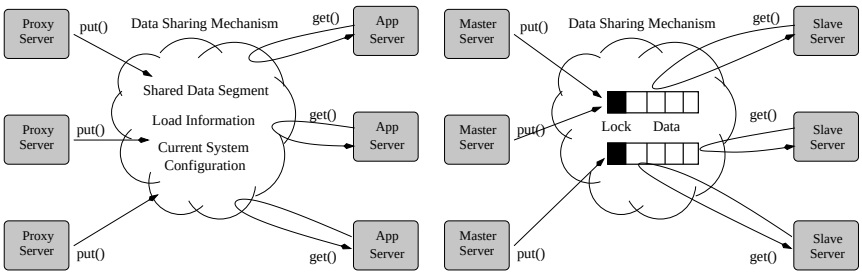


**Fig. 2.** DDSS using the proposed Framework (a) Non Coherent Distributed Data Sharing Mechanism (b) Coherent Distributed Data Sharing Mechanism

## 4 Proposed DDSS Framework and Implementation Issues

The basic idea of DDSS is to allow efficient sharing of information across the cluster by creating a logical shared memory region. It supports two basic operations, *get* operation to read the shared data segment and *put* operation to write onto the shared data segment. Figure 2a shows a simple distributed data sharing scenario with several processes (proxy servers) writing and several application servers reading certain information from the shared environment simultaneously. Figure 2b shows a mechanism where coherency becomes a requirement. In this figure, a set of master and slave servers access different portions of the shared data. All master processes waits for the lock since the shared data is currently being read by multiple slave servers.

In order to efficiently implement distributed data sharing, several components need to be built. Figure 3 shows the various components of DDSS that help in satisfying the needs of the current and next generation data-center applications. Broadly, in the figure, all the colored boxes are the components which exist today. The white boxes are the ones which need to be designed to efficiently support next-generation data-center applications. In this paper, we concentrate on the boxes with the *dashed lines* by providing either complete or partial solutions. In this section, we describe how these components take advantage of advanced networks in providing efficient services.

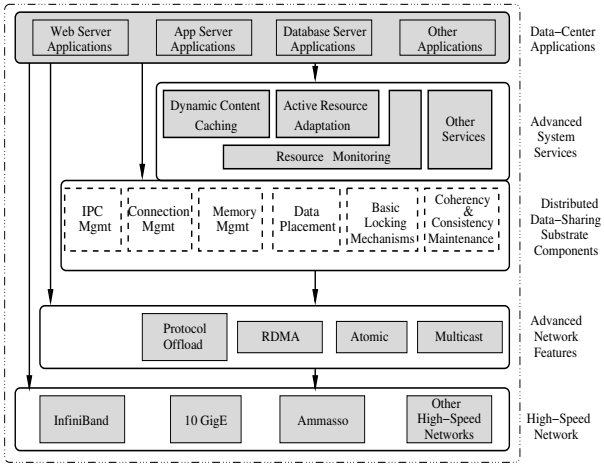


**Fig. 3.** Proposed DDSS Framework

### 4.1 IPC and Connection Management

In order to support multiple user processes or threads in a system to access the DDSS, we optionally provide a run-time daemon to handle the requests from multiple processes. We use shared memory channels and semaphores for communication and synchronization purposes between the user process and the daemon. The daemon establishes connections with other data sharing daemons and forms the distributed data sharing framework. Any service which is multi-threaded or the presence of multiple services need to utilize this component for efficient communication. Connection management takes care of establishing connections to all the nodes participating in either accessing or sharing its address space with other nodes in the system. It allows for new connections to be established and existing connections to be terminated.

### 4.2 Memory Management and Data Access

Each node in the system allocates a large pool of memory to be shared with DDSS. We perform the allocation and release operations inside this distributed memory pool. One way to implement the memory allocation is to inform all the nodes about an allocation. However, informing all the nodes may lead to large latencies.Another approach is to assign one node for each allocation (similar to home-node based approach but the node can maintain only the metadata and the actual data can be present elsewhere). This approach reduces the allocation latency. The nodes maintain a list of free blocks available within the memory pool. During a *release_ss()* operation, we inform the designated remote node for that allocation. During the next allocation, the remote node searches through the free block list and informs the free block which can fit the allocation unit. While searching for the free block, for high-performance, we get the first fit free block which can accommodate the allocation unit. High-speed networks provide one-sided operations (like RDMA read and RDMA write) that allow access to remote memory without interrupting the remote node. In our implementation, we use these operations to perform the read and write. All the applications and services mentioned in Figure 3 will need this interface in order access/update the shared data.

### 4.3 Data Placement Techniques

Though DDSS hides the placement of shared data segments, it also exposes interfaces to the application to explicitly mention the location of the shared data segment (e.g. local or remote node). For the remote nodes, the interface also allows the application to choose a specific node. In our implementation, each time a data segment is allocated, the next data segment is automatically allocated on a different node. This design allows the shared data segments to get well-distributed among the nodes in the system and accordingly help in distributing the load in accessing the shared data segments for data-center environments. This is particularly useful in reducing the contention at the NIC in the case where all the shared segment resides in one single node and several nodes needs to access different data segment residing on the same node. In addition, distributed shared segments also help in improving the performance for applications which use asynchronous operations on multiple segments distributed over the network.

### 4.4 Basic Locking Mechanisms

Locking mechanisms are provided using the atomic operations which is completely handled by modern network adapters. The atomic operations such as Fetch-and-Add and Compare-and-Swap operate on 64-bit data. The Fetch-and-Add operation performs an atomic addition at a remote node, while the Compare-and-Swap compares two 64-bit values and swaps the remote value with the data provided if the comparison succeeds. In our implementation, every allocation unit is associated with a 64-bit data which serves as a lock to access the shared data and we use the Compare-and-Swap atomic operation for acquiring and checking the status of locks. If the locks are implicit based on the coherence model, then DDSS automatically unlocks the shared segment after successful completion of *get()* and *put()* operations. Each shared data segment has an associated lock. Though we maintain the lock for each shared segment, the design allows for maintaining these locks separately. Similar to the distributed data sharing, the locks can also be distributed which can help in reducing the contention at the NIC if too many processes try to acquire different locks on the same node.

### 4.5 Coherency and Consistency Maintenance

As mentioned earlier, we support six different coherence models. We implement these models by utilizing the RDMA and atomic operations of advanced networks. However, for networks which lack atomic operations, we can easily build software-based solutions using the send/receive communication model. In the case of Null coherence model, since there is no explicit requirement of any locks, applications can directly read and write on the shared data segment. For strict, read, write coherence models, we maintain locks and *get()* and *put()* operations internally acquire locks to DDSS before accessing or modifying the shared data. The locks are acquired and released only when the application does not currently hold the lock for a particular shared segment. In the case of version-based coherence model, we maintain a 64-bit integer and use *IBV_WR_ATOMIC_FETCH_AND_ADD* atomic operation to update the version for every *put()* operation. For *get()* operation, we perform the actual data transfer only if the current version does not match with the version maintained at the remote end. In delta coherence model, we split the shared segment into memory hierarchies and support up

to *x* versions. Accordingly, applications can ask for up to *x* previous versions of the data using the *get()* and *put()* interface. Basic consistency is achieved through maintaining versions of the shared segment and applications can get a consistent view of the shared data segment by reading the most recently updated version. We plan to provide several consistency models as a part of future work.

### 4.6 DDSS Interface

Table 1 shows the current interface that is available to the end-user applications or services. The interface essentially supports six main operations for gaining access to DDSS: *allocate_ss(), get(), put(), release_ss(), acquire_lock_ss(), release_lock_ss() operations*. The *allocate_ss()* operation allows the application to allocate a chunk of memory in the shared state. This function returns a unique shared state key which can be shared among other nodes in the system for accessing the shared data. *get() and put()* operations allow applications to read and write data to the shared state and *release_ss()* operation allows the shared state framework to reuse the memory chunk for future allocations. *acquire_lock_ss() and release_lock_ss()* operations allow end-user application to gain exclusive access to the data to support user-defined coherency and consistency requirements. In addition, we also support asynchronous operations such as *async_get(), async_put(), wait_ss()* and additional locking operations such as *try_lock()* operation to support a wide range of applications to use such features.

**Table 1.** DDSS Interface

| DDSS Operation | Description |
|---|---|
| int allocate_ss(nbytes, type, ...) | allocate a block of size nbytes in the shared state |
| int release_ss(key) | free the shared data segment |
| int get(key, data, nbytes, ...) | read nbytes from the shared state and place it in data |
| int put(key, data, nbytes, ...) | write nbytes of memory to the shared state from data |
| int acquire_lock_ss(key) | lock the shared data segment |
| int release_lock_ss(key) | unlock the shared data segment |

DDSS is built as a library which can be easily integrated into distributed applications such as checkpointing, DataCutter [12], web servers, database servers, etc. For applications such as datacutter, several data sharing components can be replaced directly using the DDSS. Further, for easy sharing of keys, i.e., the key to an allocated data segment, DDSS allows special identifiers to be specified while creating the data sharing segment. Applications can create the data sharing segment using this identifier and DDSS will make sure that only one process creates the data segment and the remaining processes will get a handle to this data segment. For applications such as web servers and database servers, DDSS can be integrated as a dynamic module and all other modules can make use of the interface appropriately. In addition, DDSS can also be used to replace traditional communication such as TCP/IP. In our earlier work, cooperative caching [9], we have demonstrated the capabilities of high-performance networks for data-centers with respect to utilizing the remote memory and support caching of varying file sizes. DDSS can also be utilized in such environments. However, for very large file sizes which cannot fit in a cluster memory, applications will need to rely on the file system to store and retrieve the data. Another aspect of DDSS that is currently not supported is

fault-tolerance. This is especially required for applications such as databases. If applications can explicitly inform DDSS for taking frequent snapshots, this feature can be implemented as a part of DDSS. We plan to implement this as a part of future work.

## 5 Experimental Results

In this section, we analyze the applicability of DDSS with services such as reconfiguration and active caching and with applications such as Distributed STORM and checkpointing. We evaluate our DDSS framework on two interconnects IBA and Ammasso using the OpenFabrics implementation. The iWARP implementation of OpenFabrics over Ammasso was available only at the kernel space. We wrote a wrapper for user applications which in turn calls the kernel module to fire appropriate iWARP functions. Our experimental testbed consists of a 12 node cluster with dual Intel Xeon 3.4 GHz CPU-based EM64T systems. Each node is equipped with 1 GB of DDR400 memory. The nodes were connected with MT25128 Mellanox HCAs (SDK v1.8.0) connected through a InfiniScale MT43132 24-port completely non-blocking switch. For Ammasso experiments, we use two node dual Intel Xeon 3.0 GHz processors with a 512 kB L2 cache and a 533 MHz front side bus and 512 MB of main memory.

### 5.1 Microbenchmark

**Measuring Access Latency:** The latency test is conducted in a ping-pong fashion and the latency is derived from round-trip time. For the measuring the latency of *put()* operation, we run the test performing several *put()* operations on the same shared segment and average it over the number of iterations. Figure 4a shows the latencies of different coherence models by using the *put()* operation of DDSS using OpenFabrics over IBA through a daemon process. We observe that the 1-byte latency achieved by null and read coherence model is only 20$\mu$s and 23$\mu$s. We observed that the overhead of communicating with the daemon process is close to 10-12$\mu$s which explains the large latencies with null and read coherence models. For write and strict coherency model, the latencies are 54.3$\mu$s and 54.8$\mu$s respectively. This is due to the fact both write and strict coherency models use atomic operations to acquire the lock before updating the shared data. Version-based and delta coherence models report a latency of 37$\mu$s and 41$\mu$s respectively, since they both need to update the version status maintained at the remote node. Also, as the message size increases, we observe that the latency increases for all coherence models. We see similar trends for *get()* operations with the basic 1-byte latency of get being 25$\mu$s. Figure 4b shows the performance of *get()* operation with several clients accessing different portions from a single node. We observe that DDSS is highly scalable in such scenarios and the performance is not affected for increasing number of clients. Figure 4c shows the performance of *get()* operation with several clients accessing the same portion from a single node. Here, we observe that for relatively lesser contention-levels of up to 40%, the performance of *get()* and *put()* operations do not seem to be affected. However, for contention-levels more than 40%, the performance of clients degrades significantly in the case of strict and write coherence model mainly due to the waiting time for acquiring the lock. We see similar trends in the performance of latencies using OpenFabrics over Ammasso. We have included these results in [13].

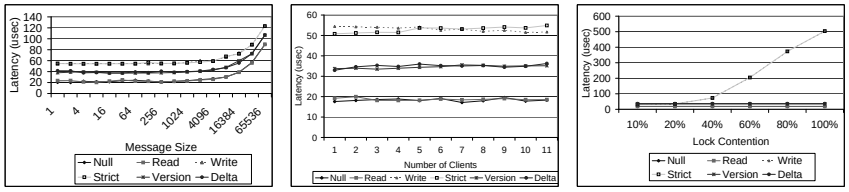


**Fig. 4.** Basic Performance using OpenFabrics over IBA: (a) *put()* operation (b) Increasing Clients accessing different portions (*get()*) (c) Contention accessing the same shared segment (*put()*)

**Measuring Substrate Overhead:** One of the critical issues to address on supporting DDSS is to minimize the overhead of the middle-ware layer for applications. We measure the overhead for different configurations (i) Direct scheme allows application to directly communicate with underlying network through DDSS library, (ii) Thread-based scheme allows application to communicate through a daemon process for accessing DDSS and (iii) Thread-based asynchronous scheme is same as thread-based scheme except that applications use asynchronous calls. We see that the overhead is less than a microsecond ($0.35\mu$s) through the direct scheme. If the run-time system needs to support multiple threads, we observe that the overhead jumps to $10\mu$s using the thread-based scheme. The reason being the overhead of round-trip communication between the application thread and the DDSS daemon consumes close to $10\mu$s. If the application uses asynchronous operations (thread-based asynchronous scheme), this overhead can be significantly reduced for large message transfers. However, in the worst case scenario, for small message sizes and scheduling of asynchronous operations followed by a wait operation can lead to an overhead of $12\mu$s. The average synchronization time observed in all the schemes is around $20\mu$s.

### 5.2 Data-Center Service Evaluation

**Dynamic Reconfiguration:** In our previous work [14] we have shown the strong potential of using the advanced features of high-speed networks in designing reconfiguration techniques. In this section, we use this technique to illustrate the overhead of using DDSS for such a service in comparison with implementations using native protocols. We modified our code base to use the DDSS and compared it with the previous implementation. Also, we emulate the loaded conditions of a real data-center scenario by firing client requests to the respective servers. As shown in Figure 5a, we see that the average reconfiguration time is only $133\mu$s for increasing loaded servers. The x-axes indicates the number of servers that are currently heavily loaded. The y-axes shows the reconfiguration time using the native protocol (white bar) and using DDSS (white bar + black bar). We observe that the DDSS overhead (black bar) is only around $3\mu$s for varying load on the servers. Also, as the number of loaded servers increase, we see no change in the reconfiguration time. This indicates that the service is highly resilient to the loaded conditions in the data-center environment. Further, we see that the number of reconfigurations increase linearly as the number of loaded servers increase from 5% to 40%. Increasing the loaded servers further does not seem to affect the reconfiguration time and when this reaches 80%, the number of reconfiguration decreases mainly due to insufficient number of free servers for performing the reconfiguration. Also, for increasing number of reconfigurations, several servers get locked and unlocked to perform efficient reconfiguration. The figure clearly shows that the contention for acquiring

locks on loaded servers does not affect the total reconfiguration time showing the scalable nature of this service. In this experiment, since we have only one process per node performing the reconfiguration, we use the direct model for integrating with the DDSS.

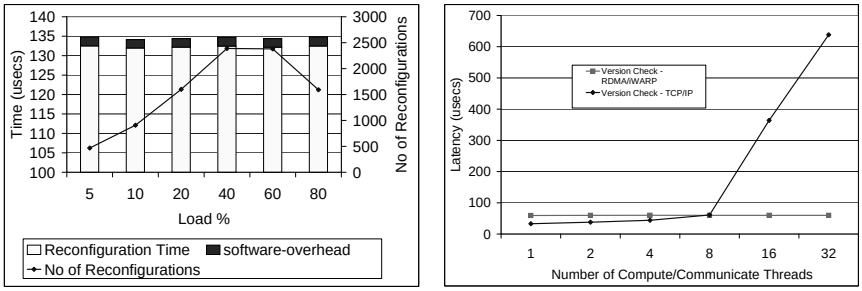


**Fig. 5.** Software Overhead on Data-Center Services (a) Active Resource Adaptation using OpenFabrics over IBA (b) Dynamic Content Caching using OpenFabrics over Ammasso

**Strong Cache Coherence:** In our previous work [7], we have shown the strong potential of using the features of modern interconnects in alleviating the issues of providing strong cache coherence with traditional implementations. In this section, we show the load resilient nature of the one-sided communication in providing such a service using DDSS over Ammasso. Figure 5b, we observe that as we increase the number of server compute threads, the time taken to check for the version increases exponentially for a two-sided communication protocol such as TCP/IP. However, since DDSS is based on one-sided operations (RDMA over iWARP in this case), we observe that the time taken for version check remains constant for increasing number of compute threads.

### 5.3 Application-level Evaluation

**STORM with DataCutter:** STORM [12] is a middle-ware service layer developed by the Department of Biomedical Informatics at The Ohio State University. It is designed to support SQL-like select queries on datasets primarily to select the data of interest and transfer the data from storage nodes to compute nodes for processing in a cluster computing environment. In distributed environments, it is common to have several STORM applications running which can act on same or different datasets serving the queries of different clients. If the same dataset is processed by multiple STORM nodes and multiple compute nodes, DDSS can help in sharing this dataset in a cluster environment so that multiple nodes can get direct access to this shared data. In our experiment, we modified the STORM application code to use DDSS in maintaining the dataset so that all nodes have direct access to the shared information. We vary the dataset size in terms of number of records and show the performance of STORM with and without DDSS. Since larger datasets showed inconsistent values, we performed the experiments on small datasets and we flush the file system cache to show the benefits of maintaining this dataset on other nodes memory. As shown in Figure 6a, we observe that the performance of STORM is improved by around 19% for 1K, 10K and 100K record dataset sizes using DDSS in comparison with the traditional implementation.

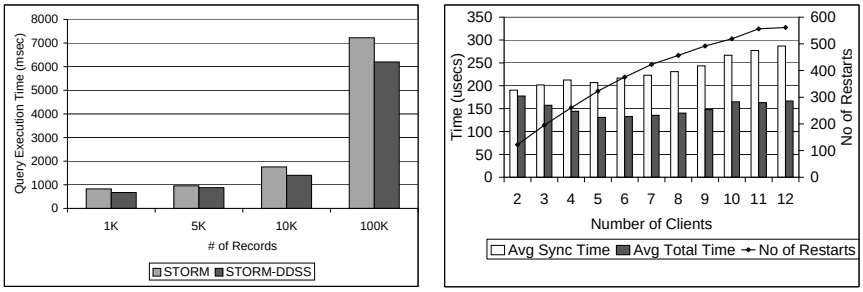


**Fig. 6.** Application Performance over IBA (a) Distributed STORM application (b) Check-pointing

**Check-pointing:** We use a check-pointing benchmark to show the scalability and the performance of using DDSS. In this experiment, every process attempts to checkpoint a particular application at random time intervals. Also, every process simulates the application restart, by attempting to reach a consistent check-point and informing all other processes to revert back to the consistent check-point at other random intervals. In Figure 6b, we observe that the average time taken for check-pointing is only around 150$\mu$s for increasing number of processes. As this value remains almost constant with increasing number of clients and application restarts, it suggests that the application scales well using DDSS. Also, we see that the average application restart time to reach a consistent checkpoint increases with the increase in the number of clients. This is expected as each process needs to get the current checkpoint version from all other processes to decide the most recent consistent checkpoint. Further, we noticed that the DDSS overhead for checkpointing in comparison with native implementation is only around 2.5$\mu$s.

## 6 Related Work

Several distributed data sharing models have been proposed in the past for a variety of environments. The key aspects that distinguish DDSS from previous work is its ability to exploit features of high-performance networks, its portability over multiple interconnects, its support for relaxed coherence protocols and its minimal overhead. Further, our work is mainly targeted for real data-center environment on very large scale clusters.

Run-time data sharing models such as InterWeave [15], Khazana [16], InterAct [17] offer benefits to applications in terms of relaxed coherency and consistency protocols. Khazana proposes the use of several consistency models. InterWeave allows users to define application-specific coherence models. Many of these models are implemented using traditional two-sided communication protocols targeting the WAN environment addressing issues such as heterogeneity, endianness, etc. Such protocols have been shown to have significant overheads in a real cluster-based data-center environment under heavy loaded conditions. Also, none of these models take advantage of high-performance networks for communication, synchronization and efficient data management. Though many of the features of high-performance networks are applicable only in a cluster environment, with the advent of advanced protocols such as iWARP included in the OpenFabrics standard, DDSS can also work well in WAN environments.

## 7 Conclusion and Future Work

This paper proposes and evaluates a low-overhead distributed data sharing substrate (DDSS) for data-center environments. Traditional data-charing implementations using ad-hoc messaging often incur high overheads and are not very scalable. DDSS

on the other hand, is designed to support efficient data management and coherence models while minimizing overheads by leveraging the features of modern interconnects. DDSS is implemented over the OpenFabrics interface and is portable across multiple interconnects including iWARP-capable networks both in LAN/WAN environments. Application-level evaluations with Distributed STORM using DDSS show close to 19% performance benefit over traditional implementation, while evaluations with check-pointing application suggest that DDSS is scalable and has a low overhead.

We plan to enhance DDSS to support advanced locking mechanisms and study the benefits of DDSS for services and applications like meta-data management, storage of BTree data structures in database servers and advanced caching techniques.

Funding Acknowledgment: This research is supported in part by Department of Energy's Grant #DE-FC02-01ER25506, and National Science Foundation's grants #CNS-0403342 and#CNS-0509452; grants from Intel, Mellanox, Sun Microsystems and Linux Networx; and equipment donations from Intel, Mellanox and Silverstorm.

## References

1. Shah, H.V., Minturn, D.B., Foong, A., McAlpine, G.L., Madukkarumukumana, R.S., Regnier, G.J.: CSP: A Novel System Architecture for Scalable Internet and Communication Services. 3rd USENIX Symposium on Internet Technologies and Systems, (2001)
2. Tang, C., Chen, D., Dwarkadas, S., Scott, M.: Integrating Remote Invocation and Distributed Shared State (2004)
3. InfiniBand Trade Association. (http://www.infinibandta.com)
4. OpenFabrics Alliance: OpenFabrics. (http://www.openfabrics.org/)
5. Shah, H.V., Pinkerton, J., Recio, R., Culley, P.: DDP over Reliable Transports (2002)
6. Ammasso, inc. (http://www.ammasso.com)
7. Narravula, S., Balaji, P., Vaidyanathan, K., Krishnamoorthy, S., Wu, J., Panda, D.K.: Supporting Strong Coherency for Active Caches in Data-Centers in InfiniBand. SAN. (2004)
8. Zhang, Y., Zheng, W.: User-level communication based cooperative caching. In ACM SIGOPS Operating Systems. (2003)
9. Narravula, S., Jin, H.W., Vaidyanathan, K., Panda, D.K.: Designing Efficient Cooperative Caching Schemes for Data-Centers over RDMA-enabled Networks. In CCGRID). (2005)
10. Chen, D., Tang, C., Sanders, B., Dwarkadas, S., Scott, M.: Exploiting high-level coherence information to optimize distributed shared state. In Proc. of the 9th ACM Symp. on Principles and Practice of Parallel Programming. (2003)
11. Vaidyanathan, K., Jin, H.W., Panda, D.K.: Exploiting RDMA Operations for Providing Efficient Fine-Grained Resource Monitoring in Cluster-based Servers. In Workshop on Remote Direct Memory Access (RDMA): RAIT, Barcelona, Spain (2006)
12. The STORM Project at OSU BMI. (http://storm.bmi.ohio-state.edu/index.php)
13. Vaidyanathan, K., Narravula, S., Panda, D.K.: Soft Shared State Primitives for Multi-Tier Data-Center Services. Technical Report OSU-CISRC-1/06-TR06, OSU (2006)
14. Balaji, P., Vaidyanathan, K., Narravula, S., Savitha, K., Jin, H.W., Panda, D.K.: Exploiting Remote Memory Operations to Design Efficient Reconfiguration for Shared Data-Centers. In Workshop on RAIT, San Diego, CA (2004)
15. Chen, D., Dwarkadas, S., Parthasarathy, S., Pinheiro, E., Scott, M.L.: InterWeave: A Middleware System for Distributed Shared State. In LCR. (2000)
16. Carter, J., Ranganathan, A., Susarla, S.: Khazana: An Infrastructure for Building Distributed Services. In ICDCS. (1998)
17. Parthasarathy, S., Dwarkadas, S.: InterAct: Virtual Sharing for Interactive Client-Server Application. Workshop on Languages, Compilers, and Systems for Computers. (1998)