An Adaptive and Flexible Replication Mechanism for Space-based Computing

Stefan Craß, Jürgen Hirsch, eva Kühn and Vesna Sesum-Cavic
Institute of Computer Languages, Vienna University of Technology, Argentinierstr. 8, Vienna, Austria

Keywords: Replication Mechanisms, Tuple Space, Coordination Middleware, Peer-to-Peer.

Abstract: The highly dynamic nature of the Internet implies necessity for advanced communication paradigms. Large modern networks exchange data without a required central authority that previously assured easy replication to avoid a loss of data. Without central authority, it is not always obvious on which client which portion of data is persisted. This is especially the case for distributed, peer-to-peer systems like ones that are based on tuple space-based coordination middleware. In recent years, many space-based solutions have been introduced but only few of them provide a built-in replication mechanism. Also, possible replication mechanisms of these systems do not provide flexibility concerning the offering of different, configurable replication schemes, replication strategies or communication protocols. Thus, such replication mechanisms can neither be adapted nor optimized for a given use case scenario. This paper introduces an asynchronous replication mechanism for space-based computing which provides a high level of flexibility by offering multiple replication approaches and can be configured and adapted for individual scenarios. This is reached by a replication manager component which uses two plugins to control replication of space content: a native space-based and a DHT-based one, both performing asynchronous multi-master replication.

1 INTRODUCTION

The classical client-server paradigm is the usual way of communication between computers across the Internet, but it implies severe problems as the server is a single point of failure. If a huge number of clients communicate with the server, they may overload it and decrease the performance of the entire system. Peer-to-Peer (P2P) networks solve this problem as each peer works as client and server at the same time, connects to other peers to request or transmit data and may dynamically join and leave the network. Beside the advantages of P2P networks like flexibility and a certain level of self-organization, they face problems of increased complexity like lacking a central register describing which information resides on which client or which clients are currently connected to the network. Once a peer leaves the network, its data is not available anymore and the only solution is to replicate each peer's data to other peers in the network.

In a distributed environment that enables clients to read and update generic data, replication is beneficial in two ways (Cecchet et al., 2008): Firstly, it helps to improve the scalability of a system as read access can be split among multiple replicas. Secondly, availability is increased as data is kept redundantly at multiple sites in order to provide fault tolerance. However, replication also induces an overhead to synchronize replicas and keep them in a consistent state. Increasing the number of replicas also increases the management effort to ensure consistency. The situation becomes even more complex if an update operation fails on certain peers. In such a case error handling must be performed to decide if the overall operation was successful or not. In the worst case the operation has to be undone on all peers. According to the CAP theorem (Gilbert and Lynch, 2002), a distributed system can, at any time, only provide two out of the three properties consistency, availability, and partition tolerance in an optimal way. Thus, if all replicas always have to be in the same, consistent state and lost messages or replica crashes occur, concurrently evaluated requests on different replicas either fail or block until the connection is restored.

Space-based middleware (Mordinyi et al., 2010) provides an architectural style for distributed processes to collaborate in a decoupled way via a shared data space. This paradigm is based on the

Craß S., Hirsch J., Kühn E. and Sesum-Cavic V..
An Adaptive and Flexible Replication Mechanism for Space-based Computing.
DOI: 10.5220/0004492505990606
In *Proceedings of the 8th International Joint Conference on Software Technologies* (ICSOFT-PT-2013), pages 599-606
ISBN: 978-989-8565-68-6

Linda tuple space model (Gelernter, 1985), which enables participants to write data tuples into a space and retrieve them using a query mechanism based on template matching. Tuple spaces can be used to synchronize independent processes via blocking queries that return their result as soon as a matching tuple is provided by another process. The XVSM (eXtensible Virtual Shared Memory) middleware model (Craß et al., 2009) adheres to this space-based computing style via space containers that are identified via a URI and support configurable coordination laws for writing and selecting data entries. Processes that access a container may write, read, or take (i.e. read and delete) entries, which generalize the tuple concept, using configurable coordination mechanisms like key-based access, FIFO queues, or template matching. Depending on the used coordination mechanism, queries for read and take operations include parameters like the key of a searched entry or the count of entries that shall be returned in FIFO order. If no matching result exists, the query blocks until it is fulfilled or a given timeout is reached, which enables decoupled communication.

If many distributed processes interact, a single space may form a performance bottleneck that hinders scalability. Thus, replicated spaces would enable scalable P2P-based solutions, but currently only a few space-based middleware systems provide built-in replication mechanisms. However, even with those that support replication, the problem is that they usually assume a fixed mechanism, but there is not one optimal replication mechanism that serves all applications equally well. The trade-off between consistency, availability and partition tolerance must be negotiated for each use case. A replication mechanism should therefore offer different replication strategies that can be configured by the user and therefore adapted to a given scenario.

In this paper, we investigate the Java-based open source implementation of XVSM, termed MozartSpaces (available at *www.mozartspaces.org*), for which we will present a flexible replication framework. We suggest an asynchronous mechanism that offers multiple replication approaches and can be configured and adapted for each scenario. A flexible plugin approach means that different replication algorithms exist, and it is easily possible to add new ones and to exchange them.

A motivating use case can be found in the domain of traffic management for road or rail networks, were nodes are placed along the track to collect data from passing vehicles and inform them about relevant events (like congestions). As nodes may fail, data must be replicated to prevent data loss. For scalability reasons, a P2P-based approach is more feasible than a centralized architecture.

The paper is structured as follows: Section 2 is dedicated to related work. Section 3 describes the suggested space-based replication framework. As a proof-of-concept two plugins are provided to control the replication of containers: i) replication via the Distributed Hash Table (DHT) implementation Hazelcast (Hazelcast, 2012) and ii) a native replication mechanism that is bootstrapped using the space-based middleware itself. Both plugins perform asynchronous multi-master replication. Section 4 evaluates the solution and analyzes benchmark results, while Section 5 provides a conclusion.

2 RELATED WORK

Replication for databases and data-oriented middleware like tuple spaces may be achieved via synchronous or asynchronous replica updates. Synchronous replication as defined by the ROWA (Read-One-Write-All) approach (Bernstein, Hadzilacos and Goodman, 1987) forces any update operation to wait until the update has been propagated to all replicas. This scales well in a system that performs many read operations but few updates. In general, however, asynchronous replication mechanisms that use lazy update propagation increase the scalability and performance dramatically (Jiménez-Peris, et al., 2003), but this is achieved at the cost of reduced consistency guarantees and more complex error handling. Depending on the requirements of a distributed application, strict consistency models based on ACID (Atomicity, Consistency, Isolation, Durability) (Haerder and Reuter, 1983) or relaxed models like BASE (Basically Available, Soft state, Eventually consistent) (Pritchett, 2008) are more suitable for data replication. While ACID transactions always guarantee consistent replica states, BASE uses a more fault-tolerant model that allows temporarily inconsistent states. In this paper, we present a replication mechanism that supports both types of consistency models.

Replication schemes define how operations are performed on specific replicas. For a space-based approach, the master-slave and multi-master replication schemes are relevant. For master-slave replication, several slave nodes are assigned to a single master node. Read operations can be performed on any node while updates are restricted to the master node, which then propagates the

changed data to the slaves. If the master node fails, another node may be elected to be the new master. If many updates occur, the master may still become a bottleneck. In this case, a multi-master approach is more feasible, where every node may accept both read and update operations. However, an additional synchronization mechanism has to be introduced between the replicated nodes to guarantee that updates are performed in the same order on each replica. The proposed replication architecture supports both types of replication depending on the used plugin.

Effective and fault-tolerant replication for space-based middleware can be achieved by letting distributed space instances collaborate using a P2P approach. Replication frameworks require a reliable way of coordinating replicas and exchanging meta data among nodes. One way to establish such a coordination channel is via Distributed Hash Tables (Byers et al., 2003), which distribute data as key-value pairs across the P2P network according to a deterministic hash function. The hashed key serves to retrieve a specific value from the network without knowing its actual storage location. In Section 3.2, we evaluate a DHT-based replication mechanism for MozartSpaces based on the Hazelcast in-memory data grid, which provides dynamic node discovery, distributed locking and a map abstraction that transparently distributes data among several nodes in a fault-tolerant way. An alternative mechanism is based on group communication, where replicas subscribe to a specific topic, e.g. for a specific container, and are informed when a new message is published. Such a channel can be established using meta containers of the space itself, as we show with our native MozartSpaces replication approach in Section 3.3.

Several related replication mechanisms have been invented for space-based middleware: **GSpace** (Russello et al., 2005) provides a Linda tuple space where every tuple type can be assigned to a specific replication policy, like replication to a fixed number of nodes or to dynamically evaluated consumers of a certain tuple type. Using a cost evaluation function based on the current space usage, the replication policy may be changed dynamically. **DepSpace** (Bessani et al., 2008) examines Byzantine fault-tolerant replication for Linda spaces using a total order multicast protocol that works correctly if less than a third of the replicas are faulty. **Corso** (Kühn, 1994) uses a replication mechanism based on a logical P2P tree-based overlay topology of replicas, where the master copy can be dynamically reassigned to another node through a primary copy migration protocol, to allow local updates on a data field. When using the eager propagation mode, updates are pushed to all replicated locations immediately, whereas for lazy propagation, updates are pulled on-demand when the corresponding data is accessed locally. **LIME** (Picco et al., 1999) provides a flexible replication approach for tuple spaces in mobile environments. Configurable replication profiles specify in which tuples a node is interested. If a matching tuple is found among neighbouring nodes, it is automatically replicated into the local space using an asynchronous master-slave approach.

Compared to the mentioned space-based solutions, the here proposed MozartSpaces replication mechanism is able to cope with different coordination laws (label, key, queue, template matching etc.) and provides a more generic replication framework that supports the plugging of arbitrary replication mechanisms.

3 CONTAINER REPLICATION

3.1 Replication Manager

Due to their flexible coordination laws, XVSM containers are not just simple lists of data entries. For any container, each supported coordination mechanism is managed by a so-called coordinator, which stores an internal container view (e.g. a map or an index) that is updated every time when entries are written to or taken from the container. This view then determines which entries are selected by a specific read or take query. Thus, to replicate a container, the entries as well as the coordinator meta data have to be distributed to remote spaces.

The proposed replication manager for XVSM containers is termed **XRM**. It consists of:

- **Replication API:** providing methods to create and destroy containers, as well as to read, write or take entries
- **Replication Plugins:** performing the specific functionality of the API methods
- **Quality-of-Service (QoS) Thread:** running in the context of the replication manager and ensuring a configurable quality level (number of replicas, consistency, etc.)

The XRM is designed in a way to allow for high flexibility and configurability through the parameters *replicationStrategy*, *minReplicaCount*, *maxReplicaCount*, and *replicaSites*. The used replication plugin is set via the *replicationStrategy*

parameter. The minimum and maximum numbers of replicas are specified by *minReplicaCount* and *maxReplicaCount*. Available locations, i.e. instances of MozartSpaces runtimes which can be used for replication, are defined via *replicaSites*. If the current XRM instance is the first one of a cluster or the plugin manages the replica locations itself (as described in Section 3.2), this parameter can be empty. Figure 1 depicts the general replication architecture, showing a client that contacts the XRM with API calls to access replicated containers.

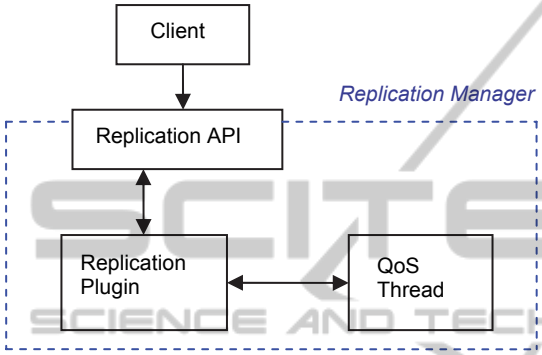

Figure 1: General replication architecture.

The replication mechanism itself is implemented in the replication plugin, to which the XRM passes the API calls. The plugin defines the basic replication mechanism, the technology for inter-process communication as well as the type of replication (synchronous or asynchronous). In this paper, a plugin using Hazelcast and a native one using MozartSpaces' own middleware mechanisms are implemented. This framework approach does not impose any limitation (by technology or operation style) and therefore, ensures flexibility.

The QoS thread keeps the number of replicas between the defined values for *minReplicaCount* and *maxReplicaCount*. It monitors all replicas and reacts to changes by triggering the creation of new replicas if the number of valid replicas drops below *minReplicaCount.* The advantage of this approach is that plugins do not need to check the status of all replicas during a method invocation, which is a time consuming operation. As the XRM and its QoS thread are running on each node of a P2P scenario, the system is able to recover from node failures as long as at least one replica of a container remains.

In order to avoid an inconsistent state between the replicas, we distinguish between **strict** and **loose consistency models**, which are applied according to the used coordinators. E.g., if on a container with five entries managed by the non-deterministic *AnyCoordinator* the same read operation is repeatedly performed, the results may differ as an arbitrary entry is selected each time. If a take operation is replicated, using this coordinator may result in replica inconsistencies, because the coordinator may delete a different entry in each replicated space container. Therefore, we use a strict consistency approach for non-deterministic coordinators, which means that when performing a take operation, nevertheless the same entries must be removed from each replica container. Therefore it is necessary to check during the execution of the take operation which entries are removed using a unique ID for each entry. These entry IDs can then be used to remove the same entries at the residual replica containers (using a key-based coordinator). The application of strict consistency ensures that replica sites are identical even after applying operations using a non-deterministic coordinator. With loose consistency, the operation is simply replicated and performed on each replica, which is sufficient for deterministic coordination mechanisms like selection by unique keys. When using loose consistency with non-deterministic coordinators, however, inconsistencies may occur.

Two types of replication meta data are considered: **Location meta data** is used to find available locations for creating new replicas or to find locations where a container was replicated. **Container meta data** consists of information related to a particular container and is used when creating new replicas. It contains the registered coordinators, the container size, coordinator meta data (e.g. keys) and the used consistency strategy. The replication meta data also has to be replicated, because without meta data it is neither possible to find other replicas, nor to create a new replica as an exact copy of an existing container.

Locking ensures consistency by avoiding concurrent operations on the same set of replicas and ensuring that updates are performed on each replica in the same order. Due to its technology dependence, locking has to be solved by the replication plugin, because there is no generic solution for this issue.

For adding new replicas, the QoS thread invokes the replication plugin to create a new container using the configuration of the container that shall be replicated. Then, already existing entries are written to this container.

For accessing entries on a replicated container, the active replication plugin retrieves the replica locations for this container. Depending on the replication semantics supported by the plugin and the used operation (read, take or write), the plugin then accesses one or more replica containers in a

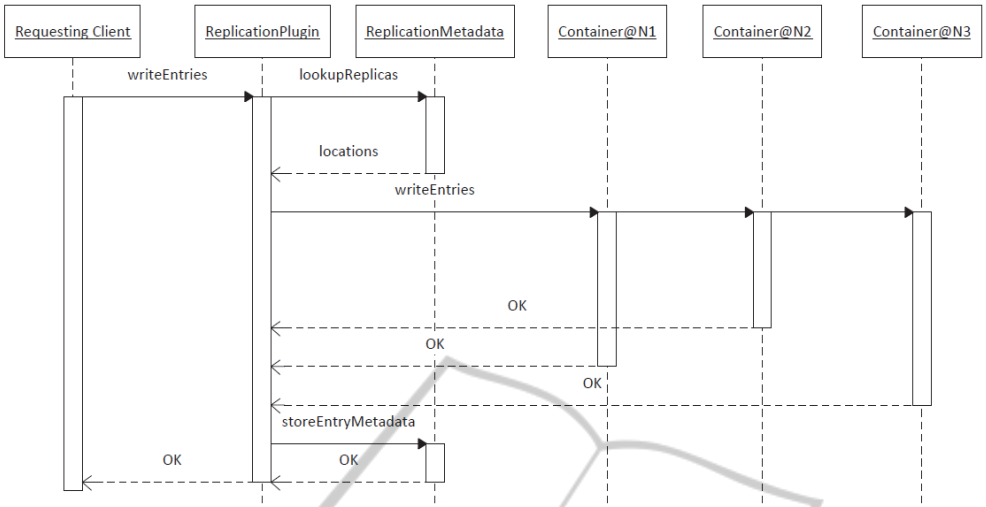

Figure 2: Sequence diagram for writing entries to a replicated container.

synchronous or asynchronous way. Figure 2 shows how new entries are written to a replicated container using an asynchronous approach. The required location and coordination meta data for this task are managed by the replication plugin.

3.2 Hazelcast Replication

The Hazelcast replication plugin uses Hazelcast (version 1.9.2.2) for handling meta data. Each portion of information stored in the Hazelcast cluster is replicated to a given number of additional cluster members and therefore the replication plugin does not need not to worry about replicating the meta data required for container replication. Also, if a new member of the Hazelcast cluster starts up, it will look for already existing members in the network neighbourhood. If such a member is found, the new member will connect to and share the meta data with the cluster. Hazelcast provides a distributed locking mechanism which acquires locks globally for all connected members in an atomic way. Therefore, both replication mechanisms (master-slave and multi-master) can be implemented. As the multi-master replication provides more flexibility and does not limit the number of write operations, this mechanism is chosen.

Figure 3 presents the architecture of the Hazelcast plugin. All instances share the same location meta data, which consists of the URIs of all available spaces and the list of replicas per container. This is realized via a Hazelcast distributed map, which replicates the meta data across a network of connected Hazelcast plugins. The location meta data is also accessible by the QoS thread to allow monitoring. The container meta data for each replicated container is stored in a similar way in the Hazelcast cluster. To access a replicated container (with name *cname*), the plugin simply retrieves the replica locations and the corresponding container meta data from the distributed storage. Entries within container *cname* are then written, read or taken using the regular MozartSpaces functions.

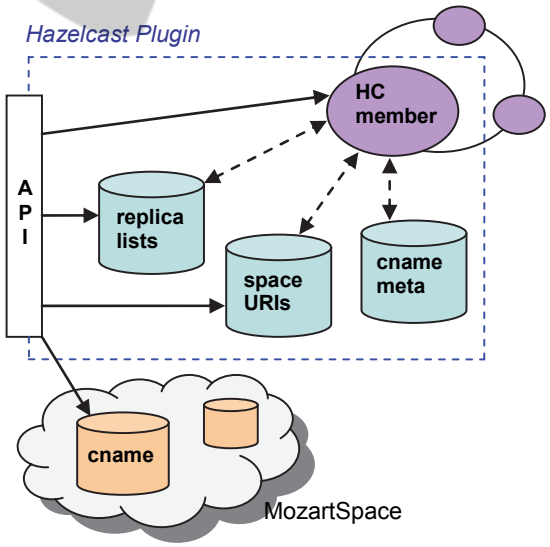

Figure 3: Hazelcast Plugin architecture (ellipses represent Hazelcast cluster members).

The plugin uses the distributed locking mechanism of Hazelcast where a lock can be acquired that is identified via the corresponding container name. Hazelcast uses a simple heartbeat approach to discover dead members, which avoids that a container is locked by a dead process and

therefore unavailable for the rest of the cluster.

3.3 Native Replication

The native replication plugin (Figure 4) does not use an additional framework for the communication between the cluster members, but only the built-in MozartSpaces functionality. To provide high flexibility and cover a high number of use cases, a multi-master replication strategy was chosen.

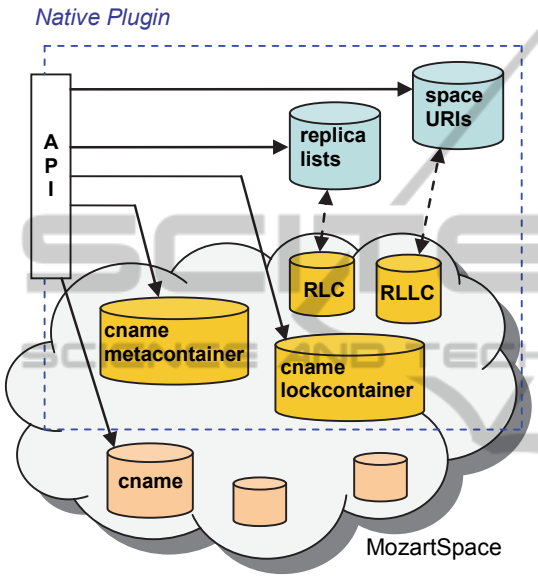

Figure 4: Native Plugin architecture.

The start-up process of the native plugin reads the initially known locations from the *replicaSites* parameter and initializes the local list of available replication locations (space URIs) in the local *ReplicationLocationLookupContainer* (RLLC). For each location, the process reads all entries from the remote *ReplicationLookupContainers* (RLC) and initializes the local replica lists in its own RLC, which contains the mapping of containers to their replica locations. Furthermore, the plugin adds its own location (URI) to the RLLC of the remote sites.

While the location meta data is managed using RLC and RLLC, the container meta data has to be stored separately. Therefore, the native plugin uses a meta container for each replicated container, which stores data like the container size, the available coordinators and the used consistency strategy. To replicate this meta container itself, a loose consistency strategy can be used to increase system performance. Strict consistency is not necessary because the meta container holds a well-known number of entries and is only accessed via deterministic key-based coordination. The meta container is added by the native plugin when creating containers on each remote peer where a replica of the requested container is created.

After each write operation on a replicated data container, the native replication plugin updates the entries' coordination data attributes (like keys and indices) in the corresponding meta container. When creating new replicas for a container, it is necessary to know all these coordinator meta data to create an exact replica of the original container.

To enable the atomic execution of multiple operations on a single space, MozartSpaces supports ACID transactions via a pessimistic locking model. However, transactional execution of updates on multiple locations requires distributed transactions. Thus, the plugin has to implement its own locking mechanism. A special lock container is created for each replicated container and registered in the corresponding meta container via its URI. If an update is performed on a replica, a lock is acquired on the lock container using MozartSpaces transactions. Because every replica of a specific container uses the same lock container, concurrent modifications can be avoided. After the update has been performed, the lock is released. If a node crashes while holding a lock, transaction timeouts ensure that the container will eventually be unlocked. If the node holding the lock container crashes, it has to be recreated at a different XRM instance.

4 EVALUATION

The performance of native and Hazelcast plugins were analyzed on a laptop with Windows 7 Enterprise 32-bit, an Intel Core 2 Duo CPU T7500 with 2.2 GHz and 4 GB RAM. The implementation was tested using the *JUnit 4* testing framework with the additional JVM parameters "*-Xmx1536m -Xms1024m*", which increase the heap space to avoid crashes when using a high number of containers and entries. For each test, the *AnyCoordinator* was used, which returns arbitrary entries with minimal overhead. The performed test cases represent two basic container operations:

- *writeEntries*: several entries are written into the containers using a single operation.
- *takeEntries*: entries are taken from the containers one by one.

Every test was performed using two instances of the same plugin to provide a basic replication environment with two cluster members. The

simulation results of performance tests using both the strict and loose consistency models are presented in Figures 5, 6, 7 and 8. Figures 5 and 6 present the results of the writeEntries and takeEntries tests using the native plugin, while Figures 7 and 8 show the results for the Hazelcast plugin. The parameter *entriesCount* represents the number of entries which are written into the container during the *writeEntries* test case and which are taken during the *takeEntries* test. All performance tests were executed with an entries count per container of 50, 100, 500, 1000, 5000 and 10000. The values are measured in milliseconds. For each replication plugin, tests with 100 and 1000 containers were performed. To eliminate variations, each test run has been performed 10 times and the mean value was used as result. Additionally, a warm-up phase was used, which allows the Java just-in-time (JIT) compiler to perform optimizations before test execution.

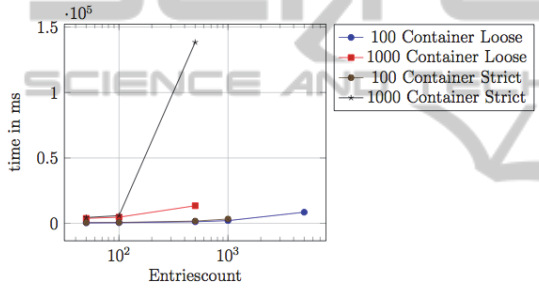

Figure 5: Native Plugin WriteEntries performance.

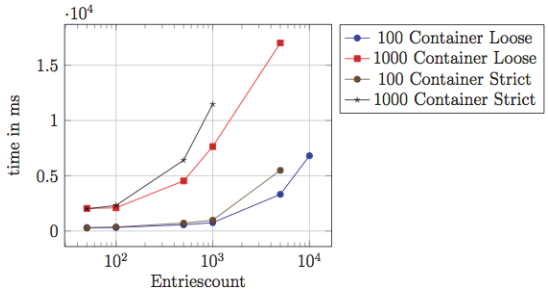

Figure 6: Native Plugin TakeEntries performance.

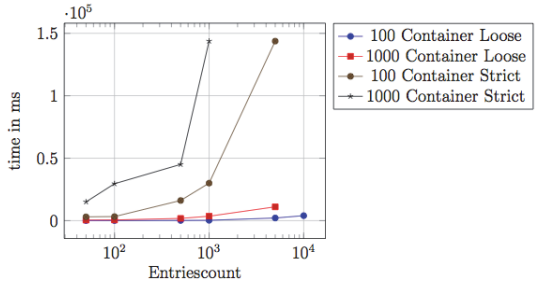

Figure 7: Hazelcast Plugin WriteEntries performance.

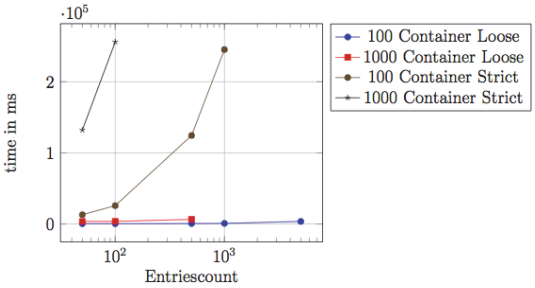

Figure 8: Hazelcast Plugin TakeEntries performance.

In summary, the native plugin performs better than the Hazelcast one, mainly due to the additional Hazelcast overhead. The Hazelcast plugin has to perform operations to the MozartSpaces cores as well as calls to the Hazelcast internals to store and retrieve meta information, and the execution of Hazelcast consumes system resources. On the other side, the native replication plugin only performs MozartSpaces calls to its underlying core and remote cores. We have also noticed that the loose consistency model scales much better than the strict model, which is expected due to the added constraints. The strict model is, however, competitive when mostly selection operations occur while using the native plugin. Additional details on the XRM implementation and the benchmarks can be found in (Hirsch, 2012).

In P2P traffic management scenarios, a suitable architecture must provide traffic information to vehicles in near-time while replicating data among nodes in a robust way to ensure fault tolerance. Asynchronous multi-master replication as provided by the presented plugins ensures that no node acts as single point of failure and that replication occurs in the background, thus preventing delays when interacting with vehicles that are only in range for a short time span. Due to the superior scalability the native replication approach appears suitable for this scenario, but the flexible framework approach allows researchers to evaluate and fine-tune further plugins that are adjusted to the specific use case.

5 CONCLUSIONS

In this paper, a customizable replication mechanism for the space-based middleware XVSM and its implementation MozartSpaces is presented. To provide a high level of flexibility, the replication manager can be configured to use several replication plugins. It is not dependent on any additional middleware for the communication layer and can be

easily extended. This way, the best replication plugin for the given use case can be chosen.

For the proof-of-concept of the flexibility of the replication manager reference implementation two plugins were implemented: a native replication plugin and a Hazelcast-based one. Hazelcast was chosen because it provides a great set of distributed data structures which can be easily used, is easy to integrate into applications, and provides an own internal replication mechanism. Both plugins perform asynchronous multi-master replication whereas the native plugin only uses functionality provided by MozartSpaces and the Hazelcast plugin uses the distributed in-memory data grid Hazelcast to store meta data. Both implementations use locks to prevent concurrent modifications of replicas. The implemented replication mechanisms focus on consistency and availability. A strict consistency model ensures that each replica contains exactly the same information. In environments where strict consistency is not needed, a loose consistency model gains more performance and scalability.

Future work will take into consideration the comparison of various intelligent replication algorithms as well as an implementation of a location-aware replication plugin based on DHTs for the mentioned traffic management use cases.

ACKNOWLEDGEMENTS

This work was partially funded by the Austrian Government via the program FFG Bridge, project LOPONODE Middleware, and by the Austrian Government and the City of Vienna within the competence centre program COMET, project ROADSAFE.

REFERENCES

Bernstein, P. A., Hadzilacos, V. and Goodman, N., 1987. *Concurrency Control and Recovery in Database Systems*. Addison-Wesley.

Bessani, A. N., Alchieri, E. P., Correia, M. and da Silva Fraga, J., 2008. DepSpace: a byzantine fault-tolerant coordination service. *ACM SIGOPS Operating Systems Review*, volume 42, 163-176. ACM.

Byers, J., Considine, J. and Mitzenmacher, M., 2003. Simple Load Balancing for Distributed Hash Tables. *Peer-to-Peer Systems II*, LNCS volume 2735, 80-87. Springer.

Cecchet, E., Candea, G. and Ailamaki, A., 2008. Middleware-based database replication: the gaps between theory and practice. In *2008 ACM SIGMOD Int'l Conf. on Management of Data*, 739-752. ACM.

Craß, S., Kühn, e. and Salzer, G., 2009. Algebraic foundation of a data model for an extensible space-based collaboration protocol. In *Int'l Database Engineering & Applications Symp.*, 301–306. ACM.

Gelernter, D., 1985. Generative communication in Linda. *ACM Trans. Program. Lang. Syst.*, 7(1):80–112. ACM.

Gilbert, S. and Lynch, N., 2002. Brewer's conjecture and the feasibility of consistent, available, partition-tolerant web services. *SIGACT News*, 33:51-59. ACM

Haerder, T. and Reuter, A., 1983. Principles of transaction-oriented database recovery. *ACM Comput. Surv.*, 15:287-317. ACM.

Hazelcast, 2012. *Hazelcast – In-Memory Data Grid*. [online] Available at http://www.hazelcast.com.

Hirsch, J., 2012. *An Adaptive and Flexible Replication Mechanism for MozartSpaces, the XVSM Reference Implementation*. Master's thesis. Vienna UT.

Jiménez-Peris, R., Patiño-Martínez, M., Alonso, G. and Kemme, B., 2003. Are Quorums an Alternative for Data Replication? *ACM Trans. Database Syst.*, 28:257-294. ACM.

Kühn, e., 1994. Fault-tolerance for communicating multidatabase transactions. In *27th Hawaii Int'l Conf. on System Sciences*, volume 2, 323-332. IEEE.

Mordinyi, R., Kühn, e. and Schatten, A., 2010. Space-based architectures as abstraction layer for distributed business applications. In *Int'l Conf. Complex, Intelligent and Software Intensive Systems*, 47–53. IEEE Computer Society.

Picco, G. P., Murphy, A. L. and Roman, G. C., 1999. LIME: Linda meets mobility. In *21st Int'l Conf. on Software Engineering*, 368-377. ACM.

Pritchett, D., 2008. BASE: An acid alternative. *Queue*, 6:48-55. ACM.

Russello, G., Chaudron, M. and van Steen, M., 2005. Dynamically adapting tuple replication for managing availability in a shared data space. *Coordination Models and Languages*, LNCS volume 3454, 109-124. Springer.