Parallel State Transfer in Object Replication Systems

Rüdiger Kapitza[1], Thomas Zeman[1], Franz J. Hauck[2], and Hans P. Reiser[3]

[1] Dept. of Computer Science 4, University of Erlangen-Nürnberg, Germany
rrkapitz@cs.fau.de, sithzema@cip.informatik.uni-erlangen.de
[2] Institute of Distributed Systems, Ulm University, Germany
franz.hauck@uni-ulm.de
[3] LASIGE, Departamento de Informática, University of Lisboa, Portugal
hans@di.fc.ul.pt

Abstract. Replication systems require a state-transfer mechanism in order to recover crashed replicas and to integrate new ones into replication groups. This paper presents and evaluates efficient techniques for parallel state transfer in such systems. These techniques enable a faster integration of replicas and improve overall service availability. On the basis of previous work on distributed download in client-server and peer-to-peer systems, we obtain parallel state-transfer mechanisms for replicated objects. Our algorithms support static and dynamic distributed download of state without a priori knowledge about the state size. A non-blocking transfer minimises the time of service unavailability during state transfer. In addition, partial state capturing is presented as an additional technique that improves the parallel transfer of large states.

1 Introduction

Replication is an established way for building reliable distributed applications. In any replication system, state transfer is required for initialising new replicas as well as for updating and recovering existing replicas. With the ongoing trend towards self-organising, dynamic distributed systems, state transfer is becoming an essential aspect of system performance and availability. For example, if the membership in a replica group changes frequently, the efficiency of the state transfer plays a non-negligible role in total system performance. In addition, synchronising the state transfer with state modification usually requires suspending the application for at least part of the duration of the transfer. This suspension time reduces system availability.

Current replication systems often use a very simple strategy for transferring the state from an available replica to the new replica. In this paper, we analyse ways to improve the performance of state transfer in replica groups. Non-blocking state transfer minimises the suspension time during the transfer, and parallel transfer from multiple state-providing replicas to a target avoids bottlenecks in the network. We evaluate the impact of various state-transfer techniques on the performance and availability of the running application.

This paper is structured as follows. The next section analyses the challenges of state transfer in object replication system and discusses related work. Section 3 presents the non-blocking and parallel variants of state transfer in our architecture. Section 4 gives a detailed experimental evaluation and Section 5 concludes.

J. Indulska and K. Raymond (Eds.): DAIS 2007, LNCS 4531, pp. 167–180, 2007.

2 Background and Related Work

The transfer of the state of an application raises the following basic questions:

- The internal application state needs to be serialised, i.e., be converted into a location-independent representation that can be transferred over the network.
- The state transfer (or, more precisely, the serialisation process) needs to be coordinated with the normal operation of the replicated application.
- The state needs to be transferred over the network.

In our prototype, the serialisation is delegated to the application. The replicated object needs to implement two methods: a `getState` method serialises the object's state into a byte stream, and a `setState` method sets the object's state on the basis of data read from a byte stream. The infrastructure provides these streams; different variants of the stream implementation can, for example, read/write directly from/to a network socket or from/to a file on a local disk. This streaming approach allows concurrency between the serialisation and the actual remote transfer, and it avoids the necessity of fully storing the serialised state. Thus, it perfectly qualifies for transferring large states.

This paper focuses on the other two questions. While most replication infrastructures need to suspend an application before state transfer and resume it afterwards, we minimise this suspension time. In addition, while current systems use a simple transfer from a single node to another, we analyse strategies for parallel state transfer from multiple up-to-date replicas to a target replica. In the following, we first discuss basic approaches to state transfer, then extend the discussion to parallel transfer mechanisms.

2.1 Basic Approaches to State Transfer

In an object replication system, the state transfer needs to be coordinated with the execution of object methods. The state has to be captured atomically, without concurrent modifications. Furthermore, the state must be captured at a specific point of time. For example, if a new replica joins a group of actively replicated objects, it needs the current state at the moment of the membership change.

The state transfer can be made in a *blocking*, *non-blocking*, and *checkpoint-based* way. Most systems support state transfer at the group-communication level. Cabaas and Mestras [1] give an overview of existing approaches to state transfer in replication frameworks, and discuss the coordination of state transfer with system operation.

In a blocking transfer, a replica resumes the execution of client requests only after the state is fully transferred to the target node. Arjuna [2] and Electra [3] support an automatic state transfer when a new member joins a replication group and block the whole system during the transfer. Phoenix [4] blocks only the members involved in the state transfer. All three systems block at least some of the group members for the complete duration of the state transfer. For large application states, this can lead to long response times [5].

In a non-blocking transfer, it is necessary only to capture the state atomically. The captured state can, for example, be stored in memory for small states, or written to hard disk for larger states. The node can resume execution while the captured state is afterwards transferred to the target over the network. Systems such as JGroups [6] and

Eternal [7] provide such a non-blocking solution. However, both systems target at the transfer of small application states that can be stored in main memory.

A checkpoint-based approach is a third variant for state transfer, used for example by Mishra et al. [8] and Castro [9]. In this approach, every replica makes periodic checkpoints and records all client requests after the last checkpoint to a log. For state transfer, the existing checkpoint and log can be transferred to the target, without the need for explicit state serialisation at the moment of state transfer.

In the domain of replicated database systems, existing work covers the recovery of replicas using the coordination support offered by group communication frameworks [10,11]. Unlike the approaches discussed above, these systems primarily target the recovery of replicas by using system properties of databases. Thus, the proposed concepts can not be directly applied to object replication systems.

This paper targets at improving and extending non-blocking as well as blocking approaches for direct state transfer in the context of object replication systems. Some of the proposed techniques can also be applied to the checkpoint-based state-transfer approach, but this is not addressed further.

2.2 Parallel Transfer

Parallel transfer of state is not popular in object replication system, but it is a standard technology in other domains such as distributed download and peer-to-peer file-sharing systems.

Rodriguez, Kirpal and Biersack [12] propose two methods for parallel download named *history-based TCP* parallel access and *dynamic TCP* parallel access. Both approaches require a dedicated unicast connection from the client to each of the providing servers. The first approach adapts the packet size depending on the available bandwidth of the accessed servers, estimated on the basis of bandwidth information gathered in earlier accesses. According to the authors, history based TCP parallel access produces good results if the network and server conditions are constant, but lead to poor performance otherwise. The dynamic TCP parallel access does not rely on potentially outdated history information. A file that is to be downloaded is divided into N blocks of equal size. The client requests a different block from every server. If a client has completely received a block, it requests a new, not yet downloaded block from that server. This simple approach assigns more blocks to faster servers, but fully loads all servers. Rodriguez et al. [13] discuss the problem that a server has an idle phase between the end of transmission of a block and the reception of a succeeding request. They suggest *request pipelining* to avoid these inter-block idle times. A new block should be requested at least one round-trip-time (RTT) before the current block is fully received.

Vazhkudai [14] proposes similar parallel access approaches, but targets at downloads of large data sets in a grid infrastructure instead of focusing on clients that access small and mid-size documents. The simplest proposed approach is *brute-force co-allocation*, in which a file is divided in n equal parts that are downloaded in parallel, with n corresponding to the number of state-providing servers. This approach takes advantage of all servers, but the time to transfer the whole file depends on the slowest connection and server. Another scheme proposed as *predictive co-allocation* corresponds to the *history based TCP* approach. Third, Vazhkudai describes two variants of a dynamic

approach that takes server and network conditions into account: *conservative load balancing* and *aggressive load balancing*. The first variant is equivalent to dynamic TCP without pipelining. The second variants uses heuristics to increase the amount of data requested from fast servers, and reduce the amount requested from slow servers or even exclude them from download altogether.

3 Decentralised State-Transfer Algorithms

In the following, we adapt the terminology of Xu et al. [15], who classify state-transfer approaches as *static-equal*, *static-unequal*, and *dynamic*. In contrast to previous work, we present an implementation that is adapted to fit the needs of distributed state transfer in active object replication. Our infrastructure provides two variants: The first variant is *static equal*, which assigns equal shares to all state-providing servers and uses small blocks to enable a continuous data flow. The second variant is *dynamic* and can be compared to dynamic TCP [12] and brute-force co-allocation [14]. In contrast to those systems, we support novel approaches for runtime optimisation that are beyond the typical mechanisms in distributed download applications.

The first issue in object replication systems is that the size of the transfer data is not known in advance. The transfer data is the result of an application-specific serialisation process, and thus it will be created "ad-hoc" at the moment the state transfer is requested. Theoretically, it is possible to first acquire the complete state from the application and then start the transfer. This is inefficient in terms of transfer time (the network transfer is delayed instead of being started in parallel to the state serialisation) and in terms of resource usage (if the state is transferred during serialisation, it is not necessary to store the full serialised state in memory or on disk). Thus, we propose algorithms that do not require the state size to be known a priori.

The second key issue is related in terms of resource usage: At the target of the state transfer, it is desirable to pass the serialised state data directly to the deserialisation process. This way, the need for storing a full copy of the serialised data in parallel to the deserialised data can be eliminated. Such functionality, however, requires that state data arrive in correct order. Some buffers for temporarily storing out-of-order data can be provided, but we want our algorithms to provide flow-control mechanisms that limit the size of such temporary storage. As a result, our approach ensures a low resource demand.

3.1 Terminology

In our system, the state data is transferred from a set of *state providers* to a single *transfer target*. The transfer protocols are defined by the exchange of *data requests* from the transfer target to state providers and *data replies* in the opposite direction. We use the following terminology:

- S is the set of state providers (servers).
- D is the state data to transfer. The size $|D|$ is not known in advance.
- A data request is defined by a tuple $< s_i, start, end >$; $s_i \epsilon S$, $start$ represents the first byte and end represent the last byte of a requested byte sequence.

– A data reply is defined as $< start, B >$, in which $start$ determines the absolute position in the state data and B represents a transmitted byte sequence, which we call a *block*.

If the requested block starts beyond the end of the state data ($start > |D|$), a state provider will indicated this fact with an empty response ($B = \{\}$). It is possible that a transfer target requests blocks beyond the end, as $|D|$ is not known in advance.

3.2 Parallel Transfer: Static Equal

The most simple strategy for distributed file transfer is *static equal*. The transfer data is split into n pieces of equal size, with n being the number of servers hosting a replica. Each replica thus has to provide a part of the state data. If the size of the transfer data is known in advance, it can easily be split into n pieces, like is done by Vazhkudai [14] and Gkantsidis et al.[16].

Without such knowledge, we must use a different approach. Each server should provide an equal amount of the state. The solution that we propose is to divide the state into small blocks of static size and to organise the transfer in rounds. In each round, the target sends a request to each server in a round-robin way, requesting a new block that has not been transferred yet. We assume that the block size $|B|$ is defined at transfer start. In round n ($n \in \mathcal{N}_0$), the requests can be constructed as follows: $< i, nr + (i-1)|B|, nr + i|B| - 1 >$, where $i = 1, ..., |S|$ designates the target of the requests. A round transfers $r = |B||S|$ of data.

As the state size is not known in advance, we define $start_{max} = +\infty$. If a response a with $D = \{\}$ is received, we compute $start_{max} = min(start_{max}, start_a)$. Now, all requests $start = nr + (i-1)|B| > start_{max}$ can be discarded. As there might be out-of-order transmissions, one has to wait for all pending requests with $start \leq start_{max}$. On the server side, the first request b that arrives with $start_b \geq |D|$ causes the server to send a response $B = \{\}$. All subsequent messages requesting data behind the end of D can be ignored. The server still has to continue participating in the state-transfer protocol, as requests for blocks before the end position might arrive out of order.

Requesting small blocks is expensive in terms of control messages, as for every block a request message has to be sent. We use *batching* to reduce the number of control messages. Instead of requesting only one block at a time, the set of all blocks of a configurable number of rounds p is requested from a state provider with a single message. Batching can easily be combined with pipelining, as suggested by Rodriguez and Biersack [13]. With pipelining, the requests for a new batch round are sent before the previous requests has fully been answered, thus reducing or eliminating the idle time of the servers between requests.

Instead of batching, two other ways might be used to reduce the cost of control messages. First, using a large block size could reduce the number of requests. Unfortunately, this strategy defeats our goal of providing a continuous stream of data that can directly be fed to the deserialisation process and thus would increase the resource usage at the receiver side. Second, the sequence of blocks could be assigned statically to each state provider at the transfer start. This way, each server would start to transfer every n-th block triggered by a single start message. This strategy leads to problems if the relative

speed of the servers differs. Again, parts of the state of very different positions might arrive at a time, requiring large buffering and thus causing resource consumption at the receiver side. Consequently, there is a need for flow control, and using explicit requests for each block (or set of blocks) automatically provides such a control mechanism.

3.3 Parallel Transfer: Static Unequal

Some existing approaches to parallel file transfer use a technique called *static unequal* by Xu et al. [15]. The difference to static equal is the addition of a phase that estimates the transfer speed from the replicas. This estimation is later used to distributed the size of the state portions that are transferred according to the relative speed. This way, faster nodes are statically assigned a larger part than slower ones.

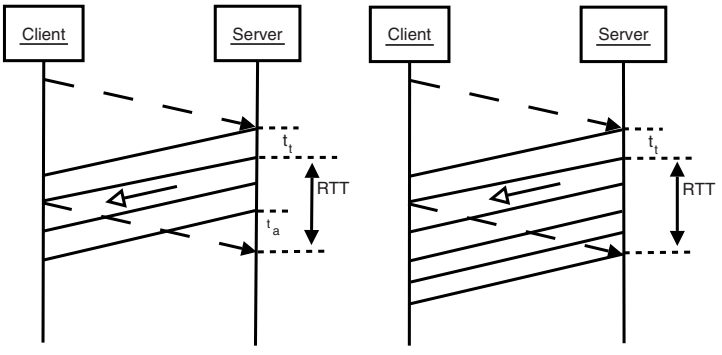
Theoretically, this principle could also be applied to parallel state transfer, using the same extensions as for static equal. The disadvantage of static unequal, however, is the addition of the estimation phase that delays the actual phase. A similar estimation can be obtained from the transfer of the first blocks in the subsequently described dynamic approach. The dynamic approach, however, is able to adjust the distribution of blocks dynamically, and, especially if flow control is used, adds no overhead compared to static unequal. Thus, we consider only the dynamic approach.

3.4 Parallel Transfer: Dynamic

While the static-equal algorithm assigns an equal part of the work to each server, *dynamic* adapts the request strategy at runtime, taking network and server condition into account. Our algorithm uses a novel approach to runtime adaptation and, in addition, introduces batching for optimisation.

The basic idea of the algorithm is to request a new block from a server each time the previously assigned block has been fully transferred. This ensures that servers which are less loaded and have a better connection (i.e., higher bandwidth and smaller round-trip time) transmit more data. As a result, the overall transfer time no longer depends on the slowest server, as it is the case for static equal. Similar to our static-equal approach, we obtain a continuous data stream with only minimal signalling overhead with a batching technique.

Our dynamic algorithm adapts the batch size individually for each server. A new batch is requested immediately after the first block has been successfully received, as shown in Figure 1. The key idea is to find an optimised batch size. If too much data is requested, this leads to bad performance in case that the state provider or the corresponding network connection slows down. On the other hand, if too few blocks (in the extreme, only one block) are requested, this causes undesirable idle times at the state provider. The ideal is to compute the batch size in a way such that a new batch request reaches the server when the last block of the previous batch has been fully transmitted. As this is not possible due to the unpredictable behaviour of the network and the server load, an estimation is used. We use a strategy inspired by Rodriguez and Biersack [13], who suggest to estimate an upper bound of the RTT and use this as a mark for submitting the next request.



(a) Sub-optimal batchsize (p=3) causing idle time

(b) Optimal batchsize (p=5)

Fig. 1. Static Dynamic request scheme with adaptive batch size

Figure 1(a) shows an idle time, t_a, that should be avoided by adjusting the batch size. As shown in Figure 1(b), the batch size should be as big as it is necessary to keep the state provider busy until the next requests arrives. The value of t_a can be computed as $t_a = RTT - t_t(p-1) = RTT - \frac{b}{C}(p-1)$. In this formula, t_t is the transfer time for a single block, b denotes the block size, p is the batch length, and C is the transfer speed of the network. In the optimal case we require $t_a = 0$, and thus we can compute $p = RTT\frac{C}{b} + 1$. The value of p depends on runtime conditions. An estimate of C can be determined by measuring the time t_t and computing $C = \frac{b}{t_t}$. The RTT can be measured in a straightforward way. As both values depend on runtime measurements that might temporarily fluctuate, an exponential moving average is used to eliminate outliers and to include previous values, but give more recent ones more impact. If the computed batch length is very short, the benefit of batching vanishes, causing a high request overhead. To compensate this fact, we introduce a configurable minimal batch length (e.g., 3).

3.5 Partial State Capturing

In a non-blocking state transfer, the serialised state data is temporarily stored at the state providers. If the state size exceeds the available memory, disk storage has to be used. Writing the state to disk is a bottleneck that limits the performance of the state acquisition, and thus also determines the period of unavailability during state serialisation. Moreover, starting the network transfer of the state in parallel to the state serialisation causes concurrent read and write operations on the same disk, which further decreases the performance.

The performance penalty of writing state data to disk can be reduced in a parallel download strategy by writing only a partial state to disk at each state provider. This requires a coordination between state capturing and state transfer. In case of the static equal approach, the parts of the state that a replica has to transmit are known at transfer

start, and thus the state acquistion process at node s_i only has to write the corresponding parts of the state, which are the blocks $s_i + n|S|$ ($n \in \mathcal{N}_0$).

Using the same approach with the dynamic transfer strategy is more difficult, as there is no fixed rule that defines the blocks that are requested from a replica. Instead, the blocks are defined at run-time. If all replicas write disjunct parts of the state, only a static equal transfer can be used. Using partial state capturing with dynamic transfer can, however, be used with a more relaxed rule. All replicas can write overlapping parts of the state (for example, by letting every replica write half of the state). The writing strategy must be defined at transfer start, and the request algorithm must take into account the availability of blocks at each state provider. The amount of overlap is a trade-off between being able to redistribute load and being able to reduce the cost of state capturing.

4 Experimental Evaluation

This section gives a brief overview of our prototype implementation and evaluates the parallel state-transfer strategies discussed in the previous section in a homogeneous LAN environment and a heterogeneous WAN setting. Finally, the impact of a non-blocking state transfer on service availability is investigated.

4.1 Implementation Overview

The proposed algorithms and mechanisms have been implemented as a protocol layer of the Java-based JGroups [6] group communication framework, which is used for replication support in our Aspectix middleware [17]. JGroups has a modular protocol stack that is configured at start-up time. An application accesses the framework via a *channel* that provides a socket-like communication endpoint. A channel provides a local unique address and enables an application to exchange unicast messages with single members and multicast messages with all members connected to the channel. Each protocol can be configured via properties during the stack initialisation. There are essentially two kinds of transmission units named *events* and *messages*. Events represent a signalling mechanism for corresponding protocol layers. Messages are application-dependent transmission units.

The message sequence diagram in Figure 2 outlines the basic signalling of the non-blocking variant of our distributed state-transfer protocol. Initially, an application requests its current state via `GET_DSTATE`. The *distributed state transfer protocol (dstp)* layer immediately returns a Java `InputStream` to the application, which uses this stream to deserialise the state. Next, the dstp layer sends a `NEED_CURRENT_STATE` message to all members including the local node. This event causes all members to enqueue all subsequent messages and a `GET_APPSTATE` message is forwarded to the replicas. This message includes a Java `OutputStream`, which the application uses to serialise the state. All members of the group that reply by sending an event named `STATE_VIEW`. If there is already an ongoing state transfer, this message and all other actions are suppressed. The joining node will be informed by `STATE_TRANSFER_DONE` that an earlier initiated state transfer has finished and can restart its state request by resending `NEED_CURRENT_STATE`. If there is no active state transfer, the requesting

node will receive the STATE_VIEW message events of all group members. Collecting these messages provides the information about all fully-functional nodes, enabling the requesting node to compute the request strategy. For example, assuming a non-blocking state transfer with partial state writing and the dynamic algorithm, not every node can provide every part of the state. Consequently, this has to be taken into account when requesting parts of the state. After reception of the STATE_VIEW message, the joining node can request the state according to the request strategy by sending dedicated DATA_REQUEST messages, which are answered by DATA_RESPONSE messages. As soon as the requesting node has received the whole state, the state transfer is finished by sending a STATE_TRANSFER_DONE.

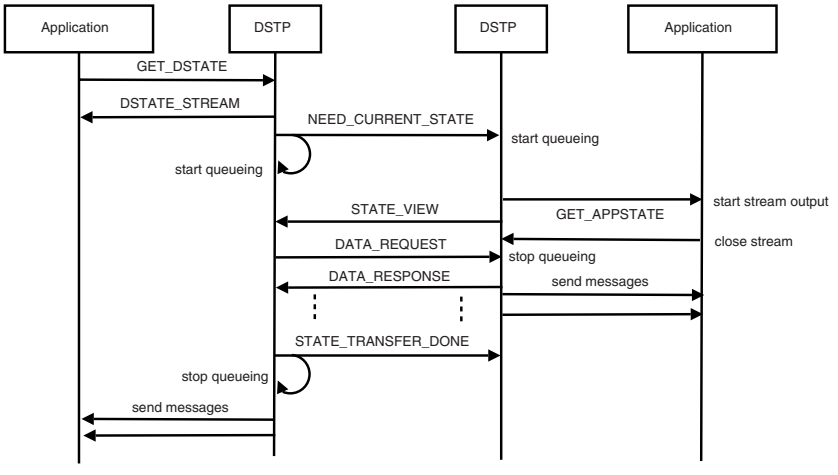


Fig. 2. Message Exchange of the Non-Blocking Distributed State Transfer Protocol

4.2 State Transfer in a Homogeneous LAN Environment

Group communication and active replication of objects often takes place in a homogeneous cluster environment. Thus, the following measurements have been made on a set of PCs with a AMD Athlon 2.0 GHz CPU and 1 GB RAM, using Linux kernel 2.6.17, SUN Java SDK 1.5.0_09, and connected by a 100 MBit/s switched Ethernet network.

We measured the time to do state transfer of state sizes between 0 and 200 MB for replication groups using the static equal and the dynamic state transfer algorithm. As the impact of parallel state transfer depends on the number of state providers, we varied the group size from one to four state-providing nodes. In all experiments, we used a fixed batch length of 10 and a block size of 16 kB. In order to compare our prototype implementations with existing state-transfer protocols, we did the same measurements with two state-transfer protocols provided by the JGroups group communication framework. The first variant, implemented by JGroups version 2.3, supports a non-blocking state transfer that requires the application to provide the state as a byte array that is transferred to the joining node. The second variant has recently been made available in

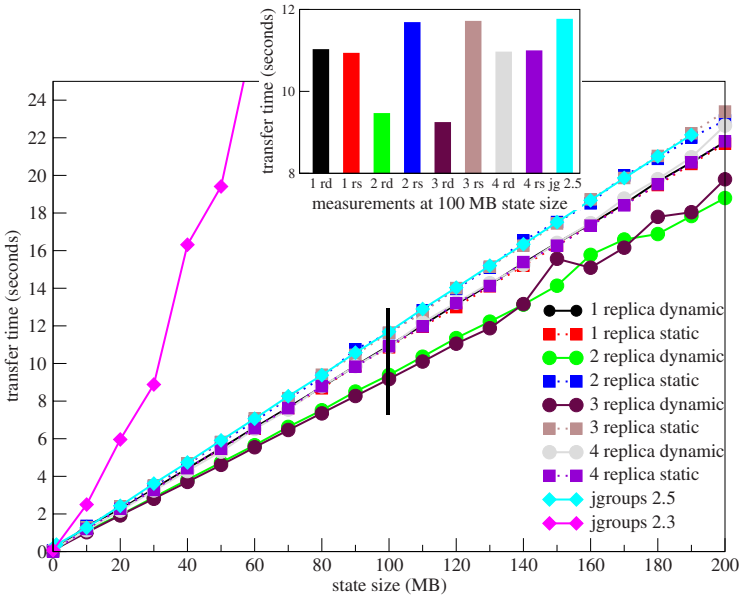


Fig. 3. State transfer in a LAN environment

the preview version of the future JGroups 2.5. It offers an API similar to our prototype and supports a blocking streaming state transfer.

Figure 3 shows the results of the measurement. The old state transfer protocol of JGroups 2.3 is not suitable for transferring states larger than 50 MB. The JGroups 2.5 state transfer protocol implementations scales better, but is not as efficient as any of our parallel state-transfer variants. The static equal parallel transfer produces very similar results for any number of state providers. The dynamic transfer offers a slight speed-up with 2 and 3 state providers, compared to only a single one. However, the performance drops back again with 4 providers. We assume that this is due to a network saturation at the link to the target and the overhead for sending requests to an increasing number of state providers.

All streaming state transfer variants produced good results that are close to each other. The dynamic variant performed slightly better than the static one, but the difference is very small. This matches our expectations, as a static equal distribution of state-transfer tasks on all nodes should be well-suited for the given homogeneous environment.

In practice, a LAN or cluster environment often is not dedicated to a single application. Thus, in a second experiment we evaluated the impact of CPU load at one of the state providing replicas. We implemented a simple load generator to produce a predictable and reproducible load. During the whole experiment, the selected node had a system CPU load between 2 and 3. We chose a group size of three replicas and a fourth node that joins the group. Again, we increased the state size from 0 to 200 MB in steps of 10 MB.

Figure 4 shows the strong impact on the state streaming Jgroups implementation. The state transfer time roughly doubles in comparison to an unloaded system. Both the dynamic and the static implementation perform better, as the state transfer is split among all state-providing replicas. The dynamic variant in general outperforms JGroups 2.5 and static equal state transfer.

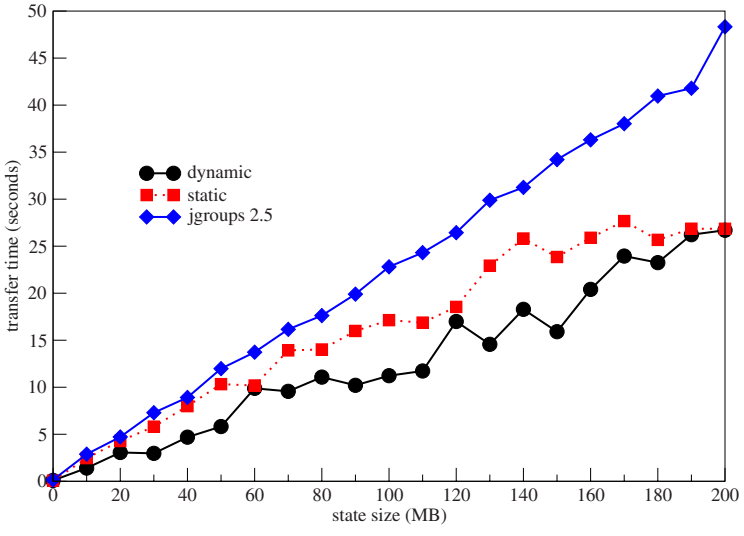


Fig. 4. State transfer in a LAN environment with load injection

In summary, the two experiments have shown that the introduction of parallel download techniques accelerates the transfer of large application states. While the benefit is only small in an idle environment, a significant speed-up is obtained in an environment with high CPU load. In both cases, the proposed dynamic state transfer algorithm outperforms the streaming state transfer offered by JGroups 2.5 and the parallel static equal algorithm.

4.3 State Transfer in a Heterogeneous WAN Environment

For evaluating the proposed techniques in a heterogeneous WAN environment, we chose a set of four different nodes. Two nodes are located in the same sub-network at the FAU Erlangen-Nuernberg, a third node *faui00a* is located in a different sub-network also at the campus of the FAU. Finally the fourth node *schirk* is located more distant at Ulm University.

In the experiment we set up a group of three replicas and let the fourth node join the group. We chose two scenarios: One time one of the machines at FAU faui00a joined the group and another time schirk the node located at Ulm University entered the group (cf. Figure 5).

Again the state transfer protocol of JGroups 2.3 did not scale and had memory problems especially when the distant node joined the group. The JGroups 2.5 protocol produced better results than the implementation of JGroups 2.3 and, as expected, requires more time for state transfer if the node at Ulm University joins the group. The transfer values of static equal are very close together, independent of the location of the joining node. Static equal is in general better than JGroups 2.5 if the distant node joins the group, but slower if the joining node is located at FAU. This is to be expected, as static equal waits for the slowest node to start another round. The dynamic parallel state transfer performs best regardless of the location of the joining node.

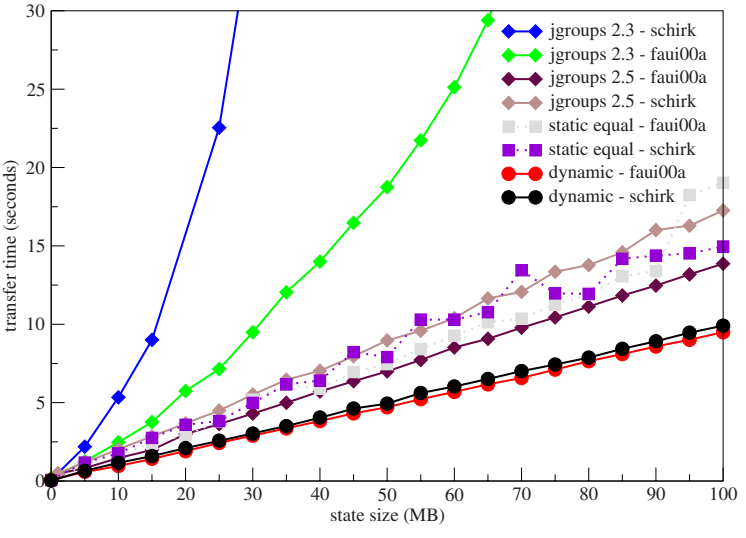


Fig. 5. State transfer in a WAN environment

4.4 Non-blocking State Transfer

This experiment does not target the reduction of the state transfer time, but instead evaluates the reduction of service unavailability caused by a state transfer.

We set up a replication group of two nodes. One node sending probe message every 100 ms to all group members. Every node that receives a probe message immediately replies to the probe and the sender records the round-trip time. Again we let a third node repeatedly join the group and raised step-wise the state size from 0 to 200 MB. During this process the joining replica recorded the time to acquire the state and the providing nodes logged the time to hand over the state to the group-communication framework. As Figure 6 details by the strong red and black lines, far less time is required to provide the state to the framework than to transfer the whole state. This is achieved as the state is temporarily saved on disk. Directly after the state provision, the application is able to respond to requests, as the second set of curves shows.

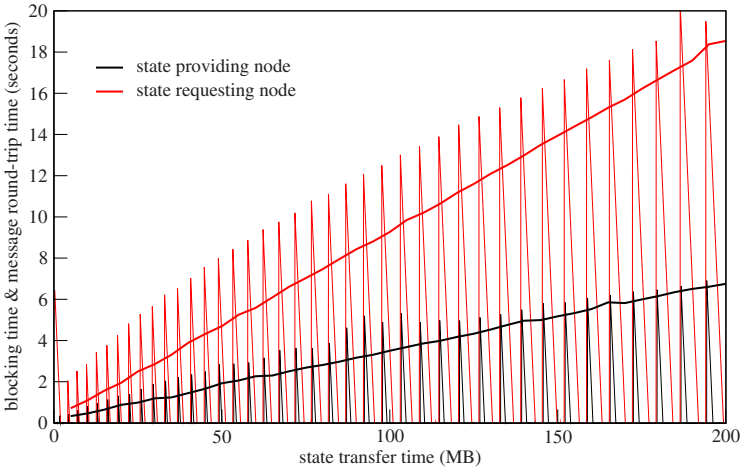


Fig. 6. The impact of a non-blocking state transfer on blocking time and message delay

5 Conclusions

This paper has presented and evaluated concepts for parallel state transfer in object replication systems. First, this paper presented and evaluated the implementation of parallel state transfer in an object replication system. While parallel download has previously been used with success in client-server systems as well as in decentralised peer-to-peer systems, it is currently not used in general infrastructures for object replication. Second, we have defined parallel state-transfer algorithms that work with an object state of unknown a priori size. In our application domain, the size of the serialised state of the replicas is usually unknown; this differs from the situation in other parallel download scenarios, in which files of known size are transferred. Third, we have presented partial state capturing as a technique that enables efficient non-blocking parallel transfer of large application states by generating only a partial state copy on disk.

An experimental evaluation has given important information about which state-transfer strategies are most important, depending on the size of the application state and the distribution of the system. We have particaluary shown that a dynamic parallel transfer enables a highly efficient state transfer. Besides minimising transfer time, our approach also minimises the time that replicas are unavailable because of suspension during state transfer.

References

1. Peña Cabañas, L., Pavón Mestras, J.: PODDP 2000 and DDEP 2000. LNCS, vol. 2023. Springer, Heidelberg (2000)
2. Parrington, G.D., Shrivastava, S.K., Wheater, S.M., Little, M.C.: The Design and Implementation of Arjuna. Computing Systems 8(2), 255–308 (1995)
3. Maffeis, S.: Adding Group Communication and Fault-Tolerance to CORBA. In: Proc. of the Conf. on Object-Oriented Technologies (Monterey, CA) USENIX, pp. 135–146 (1995)

4. Malloth, C.P.: Conception and implementation of a toolkit for building fault-tolerant distributed applications in large scale networks. PhD thesis, EPFL (1996)
5. Birman, K.: Building secure and reliable network applications. Manning Publications Co., Greenwich (1997)
6. Ban, B.: Design and implementation of a reliable group communication toolkit for Java. Technical report, Dept. of Computer Science, Cornell University (1998)
7. Narasimhan, P., Moser, L., Melliar-Smith, P.M.: State Synchronization and Recovery for Strongly Consistent Replicated CORBA Objects. In: DSN, pp. 261–270 (2001)
8. Mishra, S., Peterson, L., Schlichting, R.: Consul: a communication substrate for fault-tolerant distributed programs. Distributed Systems Engineering 1(2), 87–103 (1993)
9. Castro, M.: Practical Byzantine Fault Tolerance. Ph.D., MIT, January 2001, Also as Technical Report MIT-LCS-TR-817 (2001)
10. Kemme, B., Bartoli, A., Babaoglu, Ö.: Online Reconfiguration in Replicated Databases Based on Group Communication. In: DSN '01. Proc. of the 2001 Int. Conf. on Dependable Systems and Networks, pp. 117–130. IEEE Computer Society Press, Washington, DC, USA (2001)
11. Jiménez-Peris, R., Patiño-Martínez, M., Alonso, G.: Non-Intrusive, Parallel Recovery of Replicated Data. In: SRDS '02. Proc. of the 21st IEEE Symp. on Reliable Distributed Systems (SRDS'02), p. 150. IEEE Computer Society Press, Washington, DC, USA (2002)
12. Rodriguez, P., Kirpal, A., Biersack, E.W.: Parallel-access for mirror sites in the Internet. In: INFOCOM 2000. Nineteenth Annual Joint Conf. of the IEEE Computer and Communications Societies. Proc. IEEE, vol. 2, pp. 864–873 (2000)
13. Rodriguez, P., Biersack, E.W.: Dynamic parallel access to replicated content in the internet. IEEE/ACM Trans. Netw. 10(4), 455–465 (2002)
14. Vazhkudai, S.: Distributed Downloads of Bulk, Replicated Grid Data. J. Grid Comput. 2(1), 31–42 (2004)
15. Xu, Z., Xianliang, L., Mengshu, H., Chuan, Z.: A speed-based adaptive dynamic parallel downloading technique. SIGOPS Oper. Syst. Rev. 39(1), 63–69 (2005)
16. Gkantsidis, C., Ammar, M., Zegura, E.: On the Effect of Large-Scale Deployment of Parallel Downloading. In: WIAPP '03. Proc. of the The Third IEEE Workshop on Internet Applications, pp. 79–89. IEEE Computer Society Press, Washington, DC, USA (2003)
17. Reiser, H.P., Kapitza, R., Domaschka, J., Hauck, F.J.: Fault-tolerant replication based on fragmented objects. In: Proc. of the 6th IFIP Int. Conf. on Distributed Applications and Interoperable Systems (DAIS 2006) (2006)