Implementation in Actor Model of Leaderless Decentralized Atomic Broadcast

Alexey A. Paznikov, Andrey V. Gurin, Mikhail S. Kupriyanov
Computer engineering department
Saint Petersburg Electrotechnical University "LETI"
St. Petersburg, Russia
apaznikov@gmail.com

Abstract—**Atomic broadcast is one of the fundamental synchronization primitives for organizing shared state (consensus) in distributed systems. The promising way for implementing atomic broadcast is decentralized approach, which does not involve leader election procedure. Such approach distributes the workload uniformly among the processes, allows robustness and supports large-scale systems. Currently, there are no algorithms for atomic broadcast in actor model, which is one of the most actively developing and is widely used today. In this work, we design decentralized algorithm of atomic broadcast in the actor model. In contract to message passing model (MPI) actor model implements dynamical mapping of active entity (actor) to the threads and processes of operating system. High-level actor model allows to use developed tools for wide range of distributed applications. We have done experiments on computer cluster. The obtained latency for atomic broadcast was less than 10 ms for the subsystem of 20 nodes and broadcasting 20 requests of 100 bytes per second. In this paper, we describe the developed algorithm and the scheme for resolving system failures. The algorithm is implemented as a software library and may be used for fault-tolerant computing in distributed computer systems, for example, for implementing data replication.**

Keywords-distributed systems; consensus; atomic broadcast; leaderless atomic broadcast; actor; actor model; replication

I. INTRODUCTION

Distributed computer system (CS) is the set of elementary computers, communicating through the network for parallel execution. One of the fundamental synchronization algorithm using in CS is consensus. Consensus is the agreement among the processes in which all the participants decide the sole value. Processes in distributed CS employ consensus protocol for choosing on the values, proposed by on the processes. Consensus is the major part of the software for distributed CS which include shared state support or joint decision making for the performance of next operations.

Efficient consensus protocol must consider possible failures (errors) in distributed systems. Process may become not available for the rest of processes in the result of software or hardware errors. Consensus protocols offer different guarantees of fault tolerance for one or another kind of error. Consensus has a fundamental value for parallel programming, especially in multithreading [1-6], because it has universality property [1]: if the object x employs wait-free consensus for arbitrary number of processes, thus it can construct a wait-free implementation of any object from some number of objects x of and some number of read-write registers [1].

In the practice, consensus algorithms are used for replication – sharing object so as to ensure consistency between multiples copies of that object. Replication increase fault tolerance of the system, because if one of the processes is failed, the object remains available for users [7-10]. Replication assumes that all the copies have the same state and all the modifications of one of them are broadcasted to the rest ones. Therefore, replicated object must agree with the deterministic finite state machine model [11]: the next output value of the finite state machine is determined by its initial state and the sequence of previously executed modifications. The consensus protocol is responsible for ensuring a consistent order of receiving commands by replicas. This operation can be implemented using atomic broadcasting, which has the following properties [12]: validity, uniform agreement, uniform integrity and uniform total order. Notice, that atomic broadcast may be also used in parallel programming (e.g. in MPI) as collective communication operation for linearizable [1] modification of the state of distributed (concurrent) data structures and fault tolerance [13].

II. DECENTRALIZED ATOMIC BROADCAST

In this work, we designed the atomic broadcast algorithm in actor model based on the approach proposed in [14]. The main feature of the algorithm is that it is not based on the leader election procedure [15-19]. In the leader election, the leader becomes the bottleneck of the algorithm since it performs more operations than other processes. In addition, the failure of the leader leads to the inability of the consensus building in the whole system. The decentralized approach allows to distribute the workload between processes evenly. In addition, decentralization makes it possible to ensure the functioning of the system in case of failures of individual processes. It also allows to perform atomic broadcast in large-scale systems (each process interacts with a limited number of other processes of its local neighborhood). Decentralized approach is well established in distributed CS [20, 21]. We also use standard failure detector along with an early termination scheme.

To send messages within our approach we use overlay network. The parameters of the graph that describes it affect both the reliability of the algorithm and its performance. The vertex

The paper was prepared in Saint- Petersburg Electrotechnical University (LETI), and is supported by the Agreement № 075-11-2019-053 dated 20.11.2019 (Ministry of Science and Higher Education of the Russian Federation, in accordance with the Decree of the Government of the Russian Federation of April 9, 2010 No. 218).

978-1-7281-6949-1/20/$31.00 ©2020 IEEE

connectedness k of a graph determines how many processes must fail in order for the system to be divided into unconnected subnets. The edge connectivity λ reflects possible errors in the communication hardware. Both the vertex and edge connections are bounded by the degree d of the graph G: $k \le \lambda \le d(G)$ [22]. If $k = \lambda = d(G)$, then the graph G is considered to be optimally connected.

Within our approach each message is sended to all processes, thus diameter of the graph $D(G)$ impact on the communication time. In turn, the error diameter $D_f(G, f)$ determines the impact of server failures to the duration of a broadcast [23].

To perform a broadcast, we use a Binomial Graph (BMG) (Fig. 1) [24] and a $G_s(n, d)$-graph [25]. These graphs are suboptimally connected and have a small diameter, however, the diameter of $G_s(n, d)$ changes slightly in case of failures of some processes.

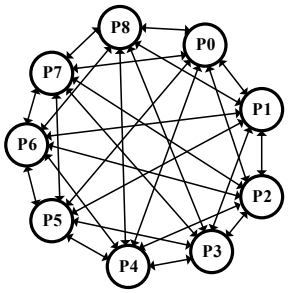

Figure 1. An example of BMG graph

To track failed processes and exclude them from the message waiting list, the algorithm includes an early termination procedure. To complete one iteration of the algorithm, processes must send their message and receive messages from all other processes. Each server processes the received notifications from failure detectors and determines which processes cannot wait for messages. A failed process could have time to send its message to neighboring processes (according to the topology): in this case, this message can still be broadcasted atomically.

III. ATOMIC BROADCAST ALGORITHM IN ACTOR MODEL

A. Actor Model in Akka.NET library

Actor model is the model parallel computing, based on the idea of actor, which is the universal primitive of parallel operations. In this work, the Akka.NET library [26, 27], which implements this model, was chosen as a tool for implementing the algorithm. Akka.NET provides scalable, distributed data processing and is currently under intensive development. In a distributed application built with Akka.NET, the main unit is the actor. Each actor contains a "mailbox" (Mailbox) – a buffer in which received messages are placed, which are then processed in FIFO order. If the actor is currently unable to process a specific message, it is possible to put it in a special buffer, Stash, from which it can be extracted and put back into the Mailbox later.

A message is sent by passing it to one of three methods: Tell, Forward, and Ask. The Tell and Forward methods are non-blocking; they send a message to another actor, without waiting for a response and return immediately. The Forward method differs in that the contents of the Sender property are specified as the sender of the message in the current context. That is, even if the message is transmitted through several intermediate actors, it will always store information about the actor who originally sent this message. The Ask method is blocking; it implements the expectation of a response from the recipient.

B. Implementation of Atomic Broadcast Algorithm in Actor model

Actors of the type ServerActor correspond the processes (servers) and implement atomic broadcast of messages. For the first start of the system, user must specify the address and port of the host, as well as the number of processes that must be created. After creation, each ServerActor is in the waiting state for initialization: it waits for the initialization message Messages.InitServer (Fig. 2, a). Messages of other types are postponed to Stash.

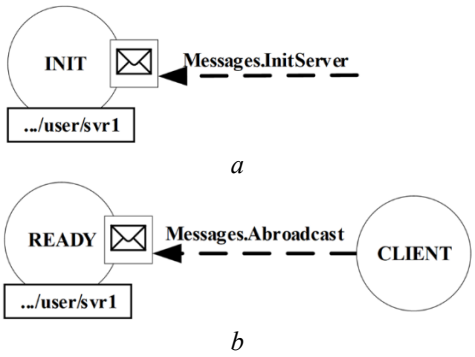

Figure 2. Client-server interaction

Messages.InitServer messages contain the data which is necessary to start the algorithm: an array IActorRef of all links to the system actors, the selected graph, the amount of information that must be displayed using the LogActor actor, the round number, as well as information about the nodes. Actors compute a topology graph (stored as adjacency lists), set of their neighbors and also start tracking failures. After initialization, the ServerActor enters the Ready state using the Become() method.

In the Ready state, the actor executes the atomic broadcast algorithm [17]. An auxiliary actor is created in the system that simulates the client's work: periodically it sends Messages.Abroadcast messages (Fig. 2, b), which should be sent atomically. Upon receiving such a message, ServerActor places it in the queue and sends it if it has not already done this in the current round. When sending, the message is placed in the Messages.Rbroadcast object. Receiving this message starts a function that checks the round number of the message, whether it was received earlier (messages are uniquely identified using Guid), places it in the M array, relays to neighbors, and also sends its own message if it did not do this in current round (Fig. 3, a). The TestTerm function is called to verify the completion of the broadcast. If this function succeeds, the messages are sorted using Guid and delivered in the same order by all servers

(Fig. 3, b), after which a new round of mailing starts if the message queue is not empty.

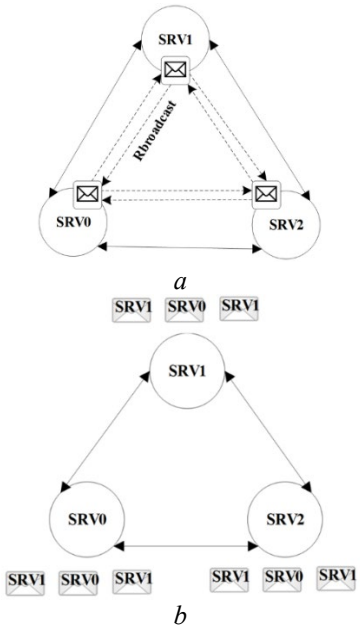

Figure 3. Communication between actors

A Terminated message is considered to be received as soon as one of the actors does not response. In response to receiving Terminated, the message is relayed and the array of tracking graphs is changed.

To add new processes, one should execute the program again, specify the address and port on which the new group of actors will be deployed, as well as the address and port of the process to which you want to join. In this case, the newly launched part of the system acts as a client; it sends a message with Messages.Abroadcast, in which the message object will be placed on Messages.MembershipRequest attachment (Fig. 4).

The request is sent atomically in the next round, after which each of the actors re-forms the topology. Then a new round begins, messages of all actors are sent, after which a special actor ACK, deployed on the node to be added, is sent a confirmation with full information about the system, which can be used to initialize the actors of this node (Fig. 5, a). As soon as added actors are initialized, they will be able to finish the round by sending their messages, and the system will continue to work as usual (Fig. 5, b).

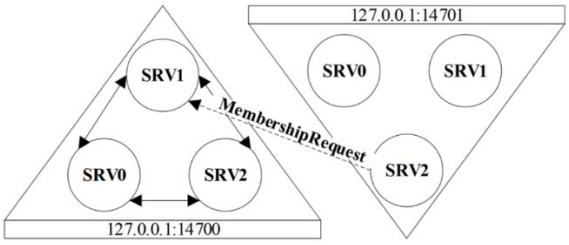

Figure 4. Sending MembershipRequest from the client

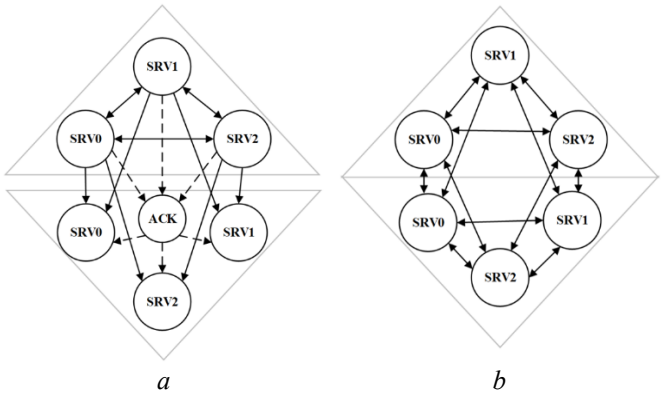

Figure 5. Initialization of a new node

IV. RESULTS AND DISCUSSION

To analyze the efficiency of the algorithm, we performed experiments on the computer cluster of the Information and Computing Center of the Novosibirsk National Research State University. Two experiments were carried out – for CS with shared memory and CS with distributed memory.

Fig. 6 shows latency values until the agreement (atomic broadcast is completed), when sending 20 requests with a frequency of 100 bytes/s, using topologies based on a binomial graph and G_s graph, for shared and distributed memory systems. For distributed memory system, with an increase in the number of servers, the synchronization time increases significantly. This is due to an increase in overhead when performing information exchanges when using an inter-site communication network. The graph G_s provide less latency than the binomial graph because it is suboptimal connected and has a smaller diameter.

Fig. 7 shows latency depending on the size of the packets in which received messages from clients. The execution time of the atomic broadcast algorithm increases with increasing data size. This is due to the increase of the execution time of the data transfer. This is especially true for subsystems with a large number of nodes. Fig. 8 shows the dependence of the delay on the frequency of receiving messages from clients. While the algorithm manages to complete the round before receiving the next batch of messages, the delays are comparable. Otherwise, messages are accumulated in the buffer until it is full.

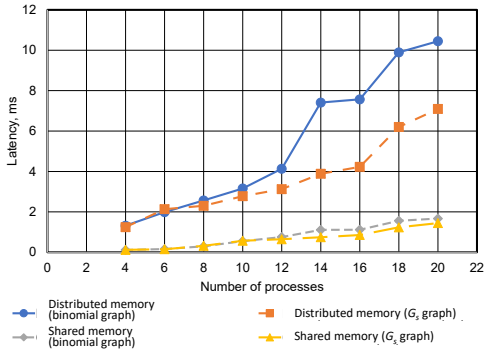

Figure 6. Latency of atomic broadcast

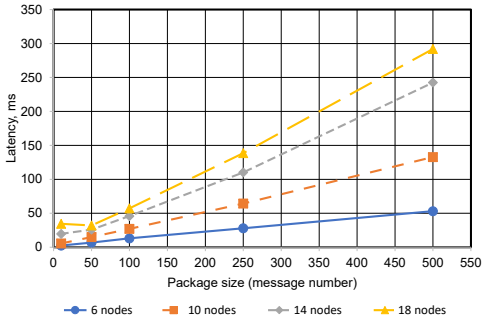

Figure 7. Latency on packet size

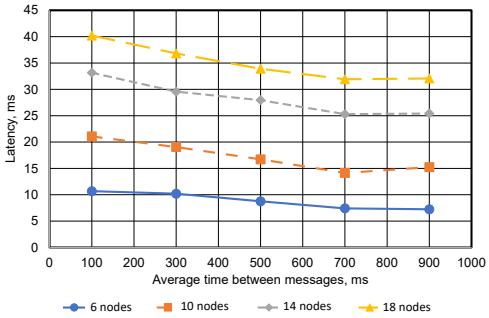

Figure 8. Latency on message rate

V. CONCLUSION

In this work, a leaderless atomic broadcast algorithm on actor model was developed. The algorithm is programmatically implemented using the Akka.NET actor library. Experiments were carried out on shared and distributed memory multiprocessor systems. The algorithm has an acceptable level of scalability. It is recommended to use the graph G_s as the topology of the system. The created tools can be used as a basis for building distributed applications in computer systems.

REFERENCES

[1] M. Herlihy, N. Shavit, "The art of multiprocessor programming," Morgan Kaufmann, 2011.

[2] A. Paznikov and Y. Shichkina, "Algorithms for Optimization of Processor and Memory Affinity for Remote Core Locking Synchronization in Multithreaded Applications," Information, vol. 9, no. 1, pp. 1-12, 2018.

[3] A. Anenkov, A. Paznikov, and M. Kurnosov, "Algorithms for access localization to objects of scalable concurrent pools based on diffracting trees in multicore computer systems," Proc. of the 14th Int. scientific-technical conference on Actual Problems of Electronic Instrument Engineering (APEIE-2018), vol. 1, part 4, pp. 374-380, 2018.

[4] A. Tabakov and A. Paznikov, "Algorithms for Optimization of Relaxed Concurrent Priority Queues in Multicore Systems," Proc. of the 2019 IEEE Conference of Russian Young Researchers in Electrical and Electronic Engineering (EIConRus), pp. 360-365, 2019.

[5] A.A. Paznikov, V.A. Smirnov, and A.R. Omelnichenko. "Towards Efficient Implementation of Concurrent Hash Tables and Search Trees Based on Software Transactional Memory," Proc. Of the 2019 International Multi-Conference on Industrial Engineering and Modern Technologies (FarEastCon), pp. 1-5, 2019.

[6] A.A. Tabakov and A.A. Paznikov. "Using relaxed concurrent data structures for contention minimization in multithreaded MPI programs," Journal of Physics: Conference Series, vol. 1399, no. 3, 2019.

[7] M. Burrows, "The Chubby lock service for loosely-coupled distributed systems," OSDI, pp. 335–350, 2006.

[8] P. Hunt, M. Konar, and F. P. Junqueira, "ZooKeeper: wait-free coordination for internet-scale systems," USENIX, pp. 145-158, 2010.

[9] B. Calder, J. Wang, A. Ogus, and N. Nilakantan, "Windows Azure Storage: A Highly Available Cloud Storage Service with Strong Consistency," SOSP, pp. 143-157, 2011.

[10] J. C. Corbett, J. Dean, M. Epstein, and A. Fikes, "Spanner: Google's globally-distributed database," USENIX, pp. 251-264, 2012.

[11] F.B. Schneider, "Implementing fault-tolerant services using the state machine approach: a tutorial," CSUR, vol. 22(4), pp. 299-319, 1990.

[12] X. Defago and A. Schiper, "Total order broadcast and multicast algorithms: Taxonomy and survey," CSUR, vol. 36, pp. 372-421, 2004.

[13] V. Zharikov, A. Paznikov, K. Pavsky, and V. Pavsky, "Adaptive Barrier Algorithm in MPI Based on Analytical Evaluations for Communication Time in the LogP Model of Parallel Computation," Proc. Of the International Multi-conference on Industrial Engineering and Modern Technologies (Far East Con-2018), pp. 1-5, 2018.

[14] M. Poke, T. Hoefler, and C. W. Glass. "AllConcur: Leaderless Concurrent Atomic Broadcast," IPDPS, pp. 205–218, 2017.

[15] P. J. Marandi, M. Primi, N. Schiper, and F. Pedone, "Ring Paxos: A high-throughput atomic broadcast protocol," DSN, pp. 527–536, 2010.

[16] D. Ongaro, and J. Ousterhout, "In search of an understandable consensus algorithm," USENIX, pp. 305–320, 2014.

[17] L. Lamport, "The part-time parliament," TOCS, vol. 16(2), pp. 133-169, 1998.

[18] L. Lamport, "Paxos made simple," SIGACT News, vol. 32, no. 4, pp. 18-25, 2001.

[19] F.P. Junqueira, B. C. Reed, and M. Serafini. "Zab: High-performance broadcast for primary-backup systems," DSN, pp. 245-256, 2011.

[20] M. G. Kurnosov and A. A. Paznikov, "Efficiency analysis of decentralized grid scheduling with job migration and replication," Proc. of ACM International Conference on Ubiquitous Information Management and Communication (IMCOM/ICUIMC), pp. 1-7, 2013.

[21] A. A. Paznikov and A. D. Anenkov. "Implementation and Analysis of Distributed Relaxed Concurrent Queues in Remote Memory Access Model," Procedia Computer Science, vol. 150, pp. 654-662, 2019.

[22] A.H. Dekker and B.D. Colbert, "Network Robustness and Graph Topology," ACSC, Dunedin. – 2004. – P. 359-368.

[23] M. S. Krishnamoorthy, and B. Krishnamurthy, "Fault diameter of interconnection networks," Computers & Mathematics with Applications. – 1987. – Vol. 13, No. 5-6. – pp. 577-582.

[24] J. Dongarra, G. Bosilca, and T. Angskun, "A Scalable and Fault-tolerant Logical Network Topology," PDCAT, pp. 471-482, 2007.

[25] T. Soneoka, M. Imase, and Y. Manabe, "Design of a d-connected digraph with a minimum number of edges and a quasiminimal diameter," Discrete Applied Mathematics, vol. 27(3), pp. 255-265, 1990.

[26] Z. Maksimovic, "Akka.NET Succinctly," Syncfusion, 121 p., 2018.

[27] A. Brown, "Reactive Applications with Akka.NET," Manning, 2019.