Robust and Strongly Consistent Distributed Storage Systems

Andria Trigeorgi(✉)

University of Cyprus, Nicosia, Cyprus
atrige01@cs.ucy.ac.cy

Abstract. The design of Distributed Storage Systems involves many challenges due to the fact that the users and storage nodes are physically dispersed. In this doctoral consortium paper, we present a framework for boosting the concurrent access to *large* shared data objects (such as files), while maintaining strong consistency guarantees. In the heart of the framework lies a fragmentation strategy, which enables different updates to occur on different fragments of the object concurrently, while ensuring that all modifications are valid.

Keywords: Distributed storage · Large objects · Concurrency · Linearizability (strong consistency)

1 Problem and Motivation

Nowadays, data are rapidly generated through applications, social media, browsers and so on. The challenge for many organizations and companies is to design an efficient storage system to cope with this data explosion. A Distributed Storage System (DSS) [17] provides data survivability and system availability. Data replication on multiple storage locations is a well known technique to cope with these issues. A main challenge due to replication, caused when the shared data are accessed concurrently, is data inconsistency.

Numerous platforms prefer high availability over consistency, due to the belief that strong consistency will burden the performance of their systems. As a result, they devise strategies to address the issue of consistency, but they rely on system coordinators to provide weaker consistency guarantees. However, modern storage systems attempt to find the balance between the consistency of the data and the availability of the system. In this work we aim to explore the development of a Robust and Strongly Consistent DSS while providing highly concurrent access to its users.

Our design is based on fundamental research in the area of distributed shared memory emulation [1,2]. These emulations provide a strong consistency guarantee, called *linearizability* (atomicity) [15], which is especially suitable for concurrent systems. Currently, such emulations, are either limited to small-size objects,

The work is supported in part by the Cyprus Research and Innovation Foundation under the grant agreement POST-DOC/0916/0090 (COLLABORATE).

S. Cherfi et al. (Eds.): RCIS 2021, LNBIP 415, pp. 670–679, 2021.
https://doi.org/10.1007/978-3-030-75018-3_51

or if two writes occur concurrently on different parts of the object, only one of them prevails. To address these limitations, we introduce a framework based on data fragmentation strategies that boost concurrency, while maintaining strong consistency guarantees, and minimize the operation latencies.

2 Existing Knowledge

In this section, we briefly review existing distributed shared memory emulations, proposed distributed file systems and discuss their strengths and limitations. Table 1 presents a comparison of the main characteristics of the distributed algorithms and storage systems that we will discuss in this document.

Attiya *et al.* [2] presents the first fault-tolerant emulation of atomic shared memory, later known as *ABD*, in an asynchronous message passing system. It implements Single-Writer Multi-Reader (SWMR) registers in an asynchronous network, provided that at least a majority of the servers do not crash. The writer completes write operations in a single round by incrementing its local timestamp and propagating the value with its new timestamp to the servers, waiting acknowledgments from a majority of them. The read operation completes in two rounds: (i) it discovers the maximum timestamp-value pair that is known to a majority of the servers, (ii) it propagates the pair to the servers, in order to ensure that a majority of them have the latest value, hence preserving atomicity. This has led to the common belief that "atomic reads must write". This belief was refuted by Dutta *et al.* [5] who showed that it is possible, under certain constraints, to complete reads in one round-trip. Several works followed, presenting different ways to achieve one-round reads (e.g., [11,12,14]). The ABD algorithm was extended by Lynch and Shvartsman [16], who present an emulation of Multi-Writer and Multi-Reader (MWMR) atomic registers.

The above works on shared memory emulations were focused on small-size objects. Fan and Lynch [7] proposed an extension, called *LDR*, that can cope with large-size objects (e.g. files). The key idea was to maintain copies of the data objects separately from their metadata; maintaining two different types of servers, replicas (that store the files) and directories (that handle metadata and essentially run a version of ABD). However, the whole object is still transmitted in every message exchanged between the clients and the replica servers. Furthermore, if two writes update different parts of the object concurrently, only one of the two prevails.

From the dozens of distributed file systems that exist in the market today, we focus on the ones that are more relevant to our work. The Google File System (GFS) [13] and the Hadoop Distributed File System (HDFS) [22] were built to handle large volume of data. These file systems store metadata in a single node and data in cluster nodes separately. Both GFS and HDFS use data stripping and replicate each chunk/block for fault tolerance. HDFS provides concurrency by restricting the file access to one writer at a time. Also, it allows users to perform only append operations at the end of the file. However, GFS supports record append at the offset of its own choosing. This operation allows multiple clients to

append data to the same file concurrently without extra locking. These systems are designed for data-intensive computing and not for normal end-users [6].

Dropbox [19] is possibly the most popular commercial DSS with big appeal to end-users. It provides eventual consistency (which is weaker than linearizability), synchronising a working object for one user at a time. Thus, in order to access the object from another machine, a user must have the up-to-date copy; this eliminates the complexity of synchronization and multiple versions of objects.

Blobseer [4] is a large-scale DSS that stores data as a long sequence of bytes called BLOB (Binary Large Object). This system uses data stripping and versioning that allow writers to continue editing in a new version without blocking other clients that use the current version. A version manager, the key component of the system, deals with the serialization of the concurrent write requests and assigns a new version number for each write operation. Unlike GFS and HDFS, Blobseer does not centralize metadata on a single machine, but it uses a centralized version manager. Thus linearizability is easy to achieve.

Our work aims in complementing the above systems by designing a highly fault-tolerant distributed storage system that provides concurrent access to its clients without sacrificing strong concurrency nor using centralized components.

Table 1. Comparative table of distributed algorithms and storage systems.

Algorithm/System	Data scalability	Data access Concurrency	Consistency guarantees	Versioning	Data Stripping
MWMR ABD [16]	NO	YES	Strong	NO	NO
LDR [7]	YES	YES	Strong	NO	NO
CoABD [8]	NO	YES	Strong	YES	NO
GFS [13]	YES	Concurrent appends	Relaxed	YES	YES
HDFS [22]	YES	Files restrict one writer at a time	Strong (centralized)	NO	YES
Dropbox [19]	YES	Creates conflicting copies	Eventual	YES	YES
Blobseer [4]	YES	YES	Strong (centralized)	YES	YES
CoBFS [our work]	YES	YES	Strong	YES	YES

3 Research Plan and Stage of Research

For the purpose of accomplishing a prototype of a Robust Distributed Storage System, with strong consistency guarantees, my research will include the following stages (quoting the estimated time for each one):

1. From existing emulations of distributed shared memory [2,7,8,10], we want to implement an optimized version of the most efficient one, in order to establish *linearizable consistency* to a unit of storage, we call a *block* (3 months).
2. Survey Data Fragmentation Strategies, focusing on block strategies. This would lead to our own fragmentation strategy, aiming to reduce the communication cost of write/read operations by splitting data into smaller atomic data objects (blocks), while enabling concurrent access to these blocks (4 months).
3. Integrate the different algorithmic modules and strategies, envisioned via a system architecture (see next section). This would entail our design and implementation framework. Based on that, we aim to introduce new consistency guarantees, that characterize the consistency of the whole object, which is composed by smaller atomic objects (blocks) (3 months).
4. Implement an erasure-coding (EC) storage (such as *ARES* [3]) which divides the object into encoded fragments and deliver each fragment to one server. The object can be recovered from a subset of the fragments. However, operations are still applied on the entire object. Thus, we can evaluate the performance of the system by combining our fragmentation approach with EC (2 months).
5. Implement a Reconfiguration service in order to mask host failures or switching between storage algorithms without service interruptions. It is expected that an existing reconfigurable algorithm (such as *ARES* [3]) may be extended to address this objective (2 months).
6. A failure prediction service can be used to estimate the risk of a device failing in order to trigger the reconfiguration service. It could be based on a monitoring service that would collect S.M.A.R.T. (Self-Monitoring, Analysis and Reporting Technology) metrics of the servers, indicating a possible drive failure. Machine Learning would be used to identify correlations on the metrics so to predict drive failures. We will integrate the failure prediction service with reconfiguration. Also, we need to specify the aggressiveness of reconfiguration: reconfiguring in every failure notification may result in many frequent reconfigurations, whereas waiting too long may disable the service due to many failures (5 months).
7. Design easy-to-use user interfaces to facilitate the use of our storage system by users that are not necessarily highly technology-trained (5 months).
8. Deploy and evaluate the system in network testbeds. An emulation testbed such as Emulab [20] will be used for developing and debugging the components of our system. However, an overlay planetary-scale testbet such as PlanetLab [23] will help us examine the performance of our system in highly-adverse, uncontrolled, real-time environments (7 months).
9. Deploy the prototype on small devices with limited computing capabilities (e.g. Raspberry Pie). During these experiments we will have the opportunity to test the durability of our storage in high concurrency conditions. Also, it will help us examine the performance of the reconfiguration operations by physically replacing the serves (5 months).

10. Further enhance our prototype implementation with more functionality and features, having the ultimate goal of developing a fully-functioning and scalable DSS, that provides high concurrency and strong consistency (5 months).

Stage of Research. I have been doing this research for the past fourteen months. During this period, items (1) to (3) and part of item (8) have been completed. We are currently working on items (4) and (5). As a proof of concept implementation, we developed a prototype implementation realizing the proposed framework [9]. Furthermore, we conducted a preliminary evaluation on Emulab [20], using Ansible Playbooks [18] for the remote execution of tasks. The evaluation shows the promise of our design; we overview the results obtained so far in the next two sections.

4 Development and Prototype Implementation

Our prototype DSS is a Distributed File System, called CoBFS, which utilizes coverable linearizable fragmented objects. The object (file), is composed of blocks, and *fragmented linearizability* [9] guarantees that all concurrent updates on different blocks are valid, and only concurrent updates on the same blocks are conflicting with each other. *Coverability* [8] extends linearizability with the additional guarantee that an update succeeds when the writer has the latest version of the object before updating it. Otherwise, an update becomes a read and returns the latest version with its associated value. The coverable version of MWMR ABD (CoABD) [8] is used as the distributed shared memory service in our system in order to ensure consistency. It allows multiple concurrent updates (writes) and reads, in an asynchronous, message-passing, crash-prone environment. We now proceed to more implementation details.

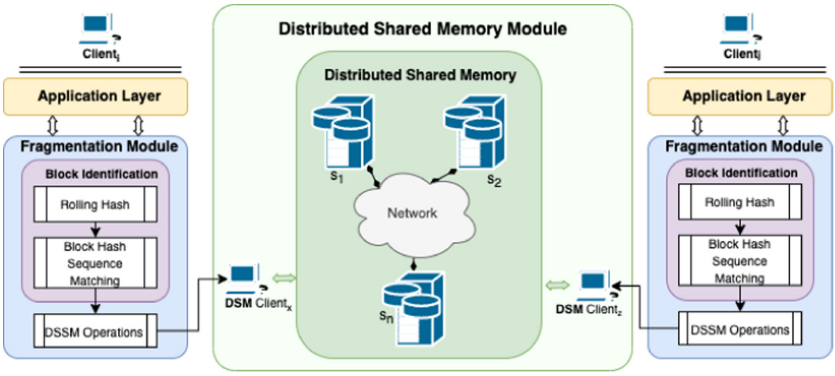

Fig. 1. The basic architecture of CoBFS

Basic Architecture Overview: The basic architecture of CoBFS, shown in Fig. 1, is composed of two main modules: (i) a Fragmentation Module (FM), and

(ii) a Distributed Shared Memory Module (DSMM). The asynchronous communication between layers is achieved by using DEALER and ROUTER sockets, from the ZeroMQ library [26]. In summary, the FM implements the fragmented object while the DSMM implements an interface to a shared memory service that allows operations on individual block objects. Following this architecture, clients can perform operations, passing commands through command-line interface. Subsequently, the FM uses the DSMM as an external service to execute these operations on the shared memory. To this respect, CoBFS is flexible enough to utilize any underlying distributed shared object algorithm.

File as a Fragmented Object: Each file f is a ***list of blocks*** with the first block being the ***genesis block*** b_{gen}, and each block having the id of its next block, whereas the last block has a null next value.

Fragmentation Module: The FM is the core concept of our implementation. Each client has a FM responsible for (*i*) fragmenting the file into blocks and identify modified or new blocks, and (*ii*) following a specific strategy to store and retrieve the file blocks from the R/W shared memory.

Update Operation: The update strategy of the FM is the most challenging part of our work. The FM uses a Block Identification (BI) module, which draws ideas from the RSYNC (Remote Sync) algorithm [24]. The BI includes three main modules, the *Block Division*, the *Block Matching* and *Block Updates*.

1. ***Block Division:*** Initially, the BI splits a given file f into data blocks based on its contents, using *rabin fingerprints* [21]. This rolling hashing algorithm performs content-based chunking by calculating and returning the fingerprints (block hashes) over a sliding window. The algorithm allows you to specify the minimum and maximum block sizes, avoiding blocks that are too small or too big. It ignores a minimum size of bytes after a block boundary is found and then starts scanning. However, if no block boundary occurs before the maximum block size, the window is treated as a block boundary. The algorithm guarantees that when a file is changed, only the hash of a modified block (and in the worst case its next one) is affected, but not the subsequent blocks.
   BI has to match each hash, generated by the rabin fingerprint from the previous step, to a block identifier.
2. ***Block Matching:*** At first, BI uses a string matching algorithm [25] to compare the current sequence of the hashes with the sequence of hashes computed in the previous file update. The string matching algorithm outputs a list of differences between the two sequences in the form of four *statuses* for all given entries: (*i*) equality, (*ii*) modified, (*iii*) inserted, (*iv*) deleted.
3. ***Block Updates:*** Based on the hash statuses, the blocks of the fragmented object are updated. In the case of equality, if a $hash_i = hash(b_j)$ then D_i is identified as the data of block b_j. In case of modification, an *update* operation is then performed to modify the data of b_j to D_i. If new hashes are inserted after the hash of block b_j, then an *update* operation is performed to create

the new blocks after b_j. The deleted one is treated as a modification that sets an empty value; thus, *no blocks are deleted*.

Read Operation: When the system receives a read request from a client, the FM issues a series of read operations on the file's blocks, starting from the genesis block and proceeding to the last block by following the next block ids. As blocks are retrieved, they are assembled in a file.

Read Optimization in DSMM: In the first phase, if a server has a smaller tag than the reader, it replies only with its tag. The reader performs the second phase only when it has a smaller tag than the one found in the first phase.

5 Preliminary Evaluation

We performed an experimental evaluation in the *Emulab* testbed [20], to compare the performance of CoBFS and of its counterpart that does not use the fragmentation module; we refer to this version as CoABD.

Experimental Setup: In our scenarios, we use three distinct sets of nodes, writers, readers and servers. Communication between the distributed nodes is via point-to-point bidirectional links implemented with a DropTail queue.

Performance Metrics: We measured performance by computing operational latency as the sum of communicational and computational delays. Additionally, the percentage of successful file update operations is calculated.

Scenarios: To evaluate the efficiency of the algorithms, we use several scenarios:

- **Scalability:** examine performance under various numbers of readers/writers
- **File Size:** examine performance when using different initial file sizes
- **Block Size**: examine performance under different block sizes (CoBFS only)

Readers and writers pick a random time between $[1..rInt]$ and $[1..wInt]$, respectively, to invoke their next operation. During all the experiments of each scenario, as the writers kept updating the file, its size increased.

Results: As a general observation, the efficiency of CoBFS is inversely affected by the number of block operations, while CoABD shows no flexibility regarding the size of the file.

Scalability: In Fig. 2(a), the update operation latency in CoBFS is always smaller than the one of CoABD, mainly because in any number of writers, each writer has to update only the affected blocks. Due to the higher percentage of successful file updates achieved by CoBFS, reads retrieve more data compared to reads in CoABD, which explains why it is slower than CoABD (Fig. 2(b)). Also, it would be interesting to examine whether the read block requests in CoBFS could be sent in parallel, reducing the overall communication delays.

File Size: As we can see in Fig. 2(c), the update latency of CoBFS remains at extremely low levels, although the file size increases. That is in contrast to the

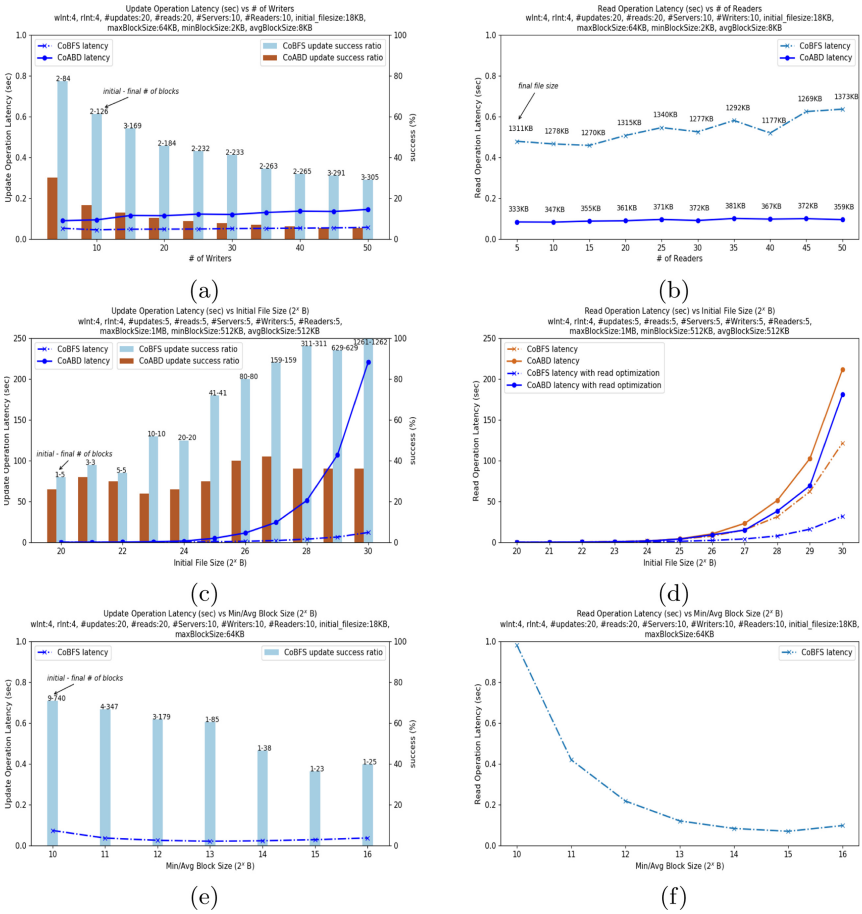

Fig. 2. Simulation results for algorithms CoABD and CoBFS.

CoABD update latency which appears to increase linearly with the file size, since it updates the whole file. The read latencies with and without the read optimization can be found in Fig. 2(d). The CoABD read latency increases sharply, even when using the read optimization. This is due to the fact that each time a new file version is discovered, CoABD sends the whole file. However, read optimization decreases significantly the CoBFS read latency, since it is more probable for a reader to already have the last version of some blocks.

Block Size: When smaller blocks are used, the update and read latencies reach their highest values and larger number of blocks (Figs. 2(e)(f)). As the minimum and average b_{sizes} increase, an update operation needs to add lower number of blocks. Similarly, smaller b_{sizes} require more read block operations to obtain the file's value. Therefore, further increase of b_{size} forces the decrease of the latencies,

reaching a plateau in both graphs. This means that the emulation finds optimal minimum and average b_{sizes} and increasing it does not give better latencies.

Concurrency: The percentage of successful file updates achieved by CoBFS are significantly higher than those of CoABD (Fig. 2(a)). As the number of writers increases (and therefore concurrency), CoABD suffers greater number of unsuccessful updates, since it manipulates the file as a whole. Also, as is shown in Fig. 2(c), a larger number of blocks yields a better success rate. The probability of two writes to collide on a single block decreases, and thus CoBFS allows more file updates to succeed in all block updates.

6 Conclusions and Future Work

We investigated several strategies in order to build CoBFS, a *Robust* and *Strongly Consistent* prototype distributed file system. This system brings forward several optimization directions and opens the path for exploring new features which can boost its reliability.

Selection of Block Size: As observed from the experiments, the operation performance is affected from the selection of minimum/maximum block sizes. Thus, it is important to have a mechanism for tuning the values for these parameters, based on the size of the file, in order to obtain the best possible performance.

Fragmentation: Due to the modular architecture of CoBFS, we can integrate other fragmentation strategies, even at the level of the shared memory module (such as EC storage mentioned in Sect. 3).

Fully-Functioning Distributed File System: We plan to evolve CoBFS into a distributed storage service handling different kind of large data, as well as impose strong security guarantees. We would also like to provide an open API, in which people could integrate their own DSSMs.

References

1. Attiya, H.: Robust simulation of shared memory: 20 years after. Bull. EATCS **100**, 99–114 (2010)
2. Attiya, H., Bar-Noy, A., Dolev, D.: Sharing memory robustly in message-passing systems. J. ACM (JACM) **42**(1), 124–142 (1995)
3. Cadambe, V., Nicolaou, N., Konwar, K.M., et al.: ARES: adaptive, reconfigurable, erasure coded, atomic storage. In: Proceedings of ICDCS, pp. 2195–2205 (2018)
4. Carpen-Amarie, A.: BlobSeer as a data-storage facility for clouds: self- adaptation, integration, evaluation. Ph.D. thesis, ENS Cachan, Rennes, France (2012)
5. Dutta, P., Guerraoui, R., Levy, R., Chakraborty, A.: How fast can a distributed atomic read be? In: Proceedings of PODC, pp. 236–245 (2004)
6. Elbert, S.T., Kouzes, R.T., Anderson, G.A., Gorton, I., Gracio, D.K.: The changing paradigm of data-intensive computing. Computer **42**, 26–34 (2009)
7. Fan, R., Lynch, N.: Efficient replication of large data objects. In: Fich, F.E. (ed.) DISC 2003. LNCS, vol. 2848, pp. 75–91. Springer, Heidelberg (2003). https://doi.org/10.1007/978-3-540-39989-6_6

8. Fernández Anta, A., Georgiou, Ch., Nicolaou, N.: CoVerability: consistent versioning in asynchronous, fail-prone, message-passing environment. In: Proceedings of NCA, pp. 224–231 (2016)
9. Fernández Anta, A., Hadjistasi, Th., Georgiou, Ch., Nicolaou, N., Stavrakis, E., Trigeorgi, A.: Fragmented objects: boosting concurrency of shared large objects. In: Proceedings of SIROCCO (2021). To appear
10. Fernández Anta, A., Hadjistasi, Th., Nicolaou, N., Popa, A., Schwarzmann, A.A: Tractable low-delay atomic memory. Distrib. Comput. **34**, 33–58 (2020)
11. Georgiou, C., Hadjistasi, T., Nicolaou, N., Schwarzmann, A.A.: Unleashing and speeding up readers in atomic object implementations. In: Podelski, A., Taïani, F. (eds.) NETYS 2018. LNCS, vol. 11028, pp. 175–190. Springer, Cham (2019). https://doi.org/10.1007/978-3-030-05529-5_12
12. Georgiou, Ch., Nicolaou, N., Shvartsman, A.A.: Fault-tolerant semifast implementations of atomic read/write registers. J. Parallel Distrib. Comput. **69**(1), 62–79 (2009)
13. Ghemawat, S., Gobioff, H., Leung, S.: The Google file system. In: Proceedings of SOSP 2003, vol. 53, no. 1, pp. 79–81 (2003)
14. Hadjistasi, T., Nicolaou, N., Schwarzmann, A.A.: Oh-RAM! one and a half round atomic memory. In: El Abbadi, A., Garbinato, B. (eds.) NETYS 2017. LNCS, vol. 10299, pp. 117–132. Springer, Cham (2017). https://doi.org/10.1007/978-3-319-59647-1_10
15. Herlihy, M.P., Wing, J.M.: Linearizability: a correctness condition for concurrent objects. ACM Trans. Program. Lang. Syst. **12**(3), 463–492 (1990)
16. Lynch, N.A., Shvartsman, A.A.: Robust emulation of shared memory using dynamic quorum-acknowledged broadcasts. In: Proceedings of FTCS, pp. 272–281 (1997)
17. Viotti, P., Vukolić, M.: Consistency in non-transactional distributed storage systems. ACM Comput. Surv. **49**, 1–34 (2016)
18. Ansible Playbooks. https://www.ansible.com/overview/how-ansible-works
19. Dropbox. https://www.dropbox.com/
20. Emulab Network Testbed. https://www.emulab.net
21. Fingerprinting. http://www.xmailserver.org/rabin.pdf
22. HDFS. https://hadoop.apache.org/docs/r1.2.1/hdfs_design.html
23. Planetlab Network Testbed. https://www.planet-lab.eu
24. RSYNC. https://rsync.samba.org/tech_report/
25. String Matching Alg. https://xlinux.nist.gov/dads/HTML/ratcliffObershelp.html
26. ZeroMQ Messaging Library. https://zeromq.org