DISCLAIMER:

This document does not meet the
current format guidelines of
the Graduate School at
The University of Texas at Austin.

It has been published for
informational use only.

The Report Committee for Jeremy J. Boyd
Certifies that this is the approved version of the following report:

Active replication vs. fusion as fault tolerance mechanisms

APPROVED BY
SUPERVISING COMMITTEE:

Supervisor:

Vijay Garg

Adnan Aziz

Active replication vs. fusion as fault tolerance mechanisms

by

Jeremy J. Boyd, B.A.

Report

Presented to the Faculty of the Graduate School of

The University of Texas at Austin

in Partial Fulfillment

of the Requirements

for the Degree of

Master of Science in Engineering

The University of Texas at Austin

May 2016

Dedication

For Lydia, whose love, support, and encouragement have been indispensable.

Abstract

Active replication vs. fusion as fault tolerance mechanisms

Jeremy J. Boyd, M.S.E.

The University of Texas at Austin, 2016

Supervisor: Vijay Garg

This report compares two strategies for crash fault tolerance of nodes in distributed systems: active replication and fusion. To tolerate f crash faults, *active replication* maintains f backup servers for each primary. *Fusion*, however, maintains a set of f backup servers that contain the replicated data for all primaries in coded form. If n primaries each contain m data to be backed up, then, active replication requires $O(nmf)$ space, while fusion requires only $O(mf)$ space. These savings come at the cost of additional time during the recovery process due to additional messages and computation. For this report, we have implemented an application in which primary nodes maintain increasingly large data structures and periodically crash. Both active replication and fusion are evaluated as recovery mechanisms for the crashed nodes. The mechanisms are evaluated for performance across three metrics: backup size, time spent during updates to the backup, and recovery time. Our results validate theoretical expectations put forward in the literature – that fusion claims significant space savings at the cost of high recovery time. In the most extreme measured case, fusion costs approximately 83% of the space that replication does, while recovery time is three orders of magnitude more expensive in

fusion (3.4s) than in replication (0.0037s). However, we also find that the gap between fusion and replication grows as nodes are introduced to the system. We find furthermore that fusion performs more consistently in update time due to the high variability of multicasting within active replication systems.

Table of Contents

List of Tables

List of Figures

Chapter 1: Introduction

In many distributed systems, the system's member nodes must maintain state in memory for correct or efficient functioning. This reliance on memory presents a problem when nodes suffer crash faults, as they often do. For quality of service, system maintainers have implemented crash fault tolerance solutions that allow a crashed node to recover its state and return to the system without disrupting the system's ordinary performance.

These solutions have generally relied on *active replication* [1, 2], in which a node's state is duplicated by some number of other nodes. When a crash fault occurs, the crashed node recovers its previous state from its replicas before re-entering the system. For tolerance of f faults among n nodes, then, a total of nf copies of state must be maintained.

A separate technique known as *fusion* [4, 5] has been proposed in which all nodes back up their state to a single set of *fused* backups. For f-fault tolerance, only f backups are required. As a result, fusion is much more space efficient than active replication. However, the recovery process requires that the backup server communicate with each non-faulty node and compute the crashed node's previous state from its fused data structure.

In this report we measure the performance of both active replication and fusion when applied within a generic model application. Each node in the application maintains some state while a large number of insert operations are performed on the application, while periodic crashes are introduced at the nodes to trigger the recovery process. We confirm that fusion achieves significant space savings at the cost of recovery time.

Updates within our fusion system are faster than in active replication, likely due to the higher number of messages required by active replication.

The structure of this report is as follows. In Chapter 2, we review active replication and fusion. Chapter 3 describes the model application as well as salient differences between the active replication and fusion versions. In Chapter 4, we present our evaluation of both mechanisms as they pertain to crash fault recovery in the model application. Finally, Chapter 5 presents some conclusions and suggestions for further work.

Chapter 2: Replication strategies

In this section we briefly review the two replication strategies under investigation in this report. Active replication offers fault tolerance with relatively simple maintenance and recovery mechanisms in exchange for a large amount of space usage. Fusion achieves space efficiency gains relative to active replication at the expense of more complex recovery time.

ACTIVE REPLICATION

In active replication [1], an in-memory data structure is maintained at several nodes or replicas, which form a group. The number of replicas required in the group is determined by the desired fault tolerance: in order to recover from f crash faults within a group, $f + 1$ replicas must be maintained. If the group comprises n nodes, this results in nf total copies of the data structure. For example, in Figure 1 we have six groups, each of which maintains an in-memory queue, and we want to protect against two crash faults for each group. Then a primary copy and two backup copies of the queue must be maintained, resulting in 18 queues in memory.

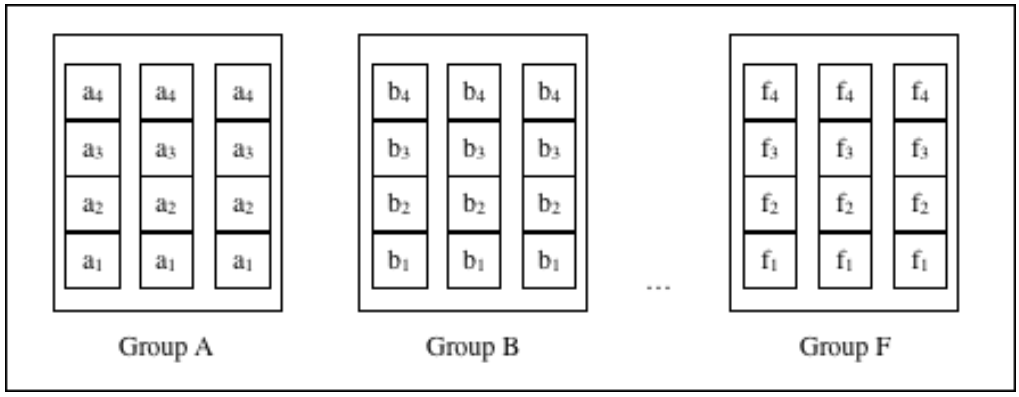

Figure 1: Replication of three queues over six groups with $f = 2$

Write operations to the replicas (e.g., updates and inserts) are typically communicated via an atomic broadcast (or multicast) protocol [3, 6]. This ensures that each node within the group eventually receives all messages and that every node processes all messages in the same order [2]. There is therefore no risk of any node failing to receive a write or acting on updating messages in a way inconsistent with other members of the group. However, the atomicity requirement can be relaxed for "read" messages – i.e., messages that do not change data but only request a view of it. In this case, read operations may not be fully up-to-date, but they are consistent with the state of the replicated data structure at some point in time.

Recovery within active replication systems is straightforward. A crashed node r_c may re-enter a group once it has received the correct state from a non-faulty node [2]. The correct state depends on the system's configuration and may be the state from just prior to r_c's crash or a state resulting from messages sent after r_c's crash. We discuss our example system's configuration and re-entry procedure in the next chapter.

Active replication is a widely used and thoroughly developed mechanism for fault tolerance. For a broader treatment of multicast algorithms and their application to replication, the reader is referred to [15] and [8], respectively. The use of replication in fault-tolerant services is discussed in detail at [2] as well as [13, 14].

FUSION

Introduction

In a fusion-based system, the data structures for all n nodes are *fused* at the backup [4, 5]. Recovery for a fused node is accomplished by querying the non-fused value from every other node whose data is represented in the fused data structure and

performing the inverse of the fusion operation on the fused data structure. In this respect, fusion is conceptually similar to RAID5 [20] and erasure coding.

To illustrate fusing simple pieces of data, suppose we have a group of six nodes, *a* through *f*, each of which has a single bit *b* to back up. Assign bits of alternating value to each node so that we have $b_a = 0$, $b_b = 1$, $b_c = 0$, etc. The fusion operator for bits is XOR. Applied bitwise, this results in fused bit at the backup $b_{fused} = 1$. To recover a particular node's bit, say b_c, we need only perform the XOR operation with the individual bits b_a through b_f. A similar strategy can be pursued for integers, using addition and subtraction as the fusion operator and its inverse, respectively.

Fusing complex data structures

For more complex structures, fusion employs a strategy of maintaining an auxiliary data structure at each primary node representing the placement of each element of its data structure at the backup. The space taken by this auxiliary structure is constant relative to the original structure at the primary. Similar structures exist at the backup to track the order of elements at the primary nodes. Figures 2-4 illustrate fusion for a linked list, the nodes of which contain bits as their data.

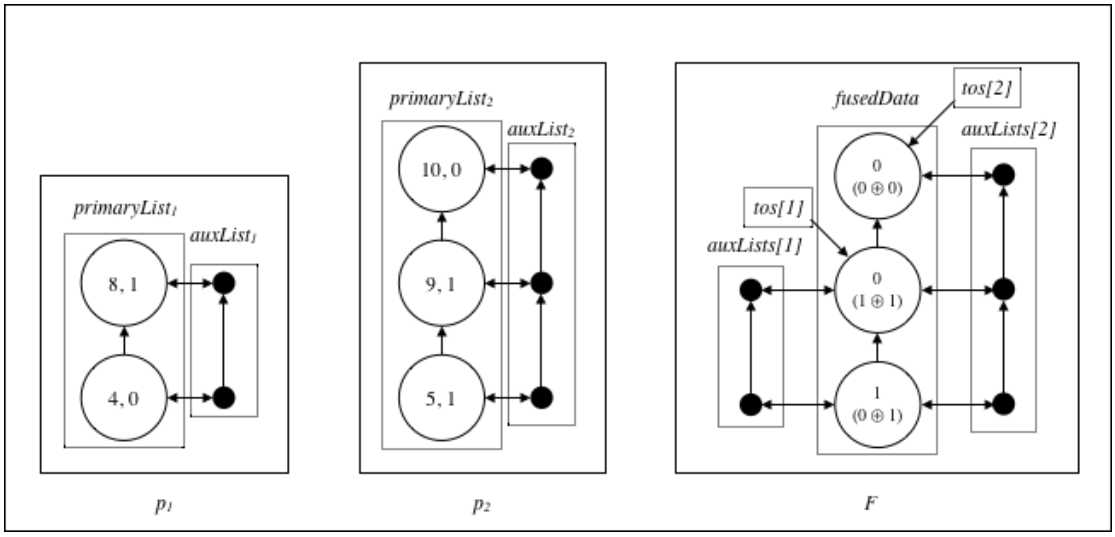

Figure 2: Two primary linked lists and their fusion

In Figure 2, each primary p_i maintains a linked list $auxList_i$ mirroring the order of its nodes in the linked list at the backup F. The backup maintains two data structures in addition to the fused linked list itself: *auxLists*, a collection of linked lists which represent the ordering of the elements at each primary, and *tos*, a collection of pointers to the node last used by each primary in the fused linked list.

Inserts are handled differently depending on whether we are adding a new node at the primary. When a client requests an insert of (k, v) at p_i, the primary first checks whether k is already in $primaryList_i$. If it is, the primary first gets the original value v' for index k, then updates the node to contain the new value. Finally, it sends a message $insert(k, v, v')$ to F. Upon receiving a message $insert(k, v_i, w_i)$ from a primary p_i, F first looks up the auxiliary structure *auxLists[i]*. If k already exists in *auxLists[i]*, then that node's value is updated with $v_i \oplus w_i$. Otherwise, we are adding a new node for the primary. Figure 3 illustrates this case.

In Figure 3, a new node for $primaryList_1$ is created with a value of 1, and an auxiliary node is created for $auxList_1$ with the new primary node as its value. A pointer to

the auxiliary node is added to the primary node so that they now reference each other. The primary then appends the auxiliary node to $auxList_1$ and inserts the primary node into the linked list normally. Finally, the primary sends F an insert message – in this case, *insert*(5, 1, 0).

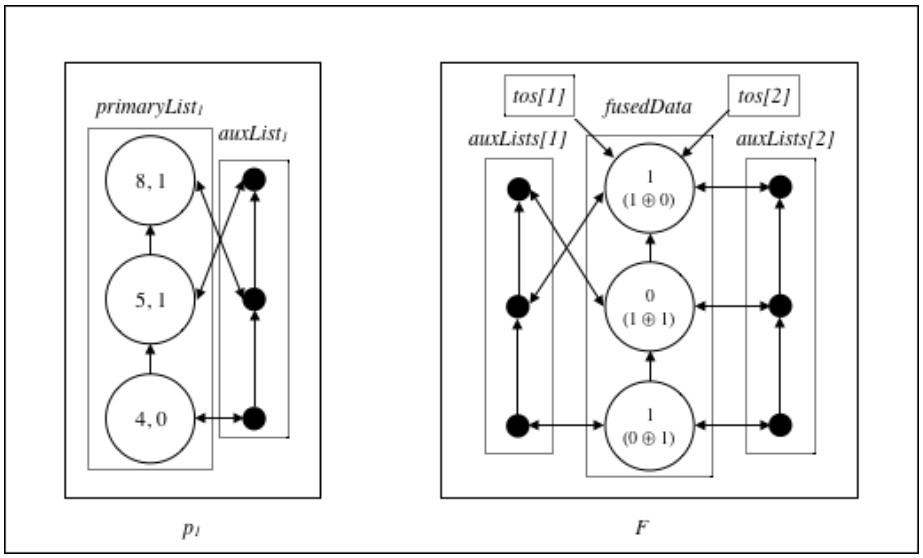

Figure 3: p_1 and F after *insert*(5, 1)

Since we already have three nodes at the backup, the final node of *fusedData* is updated with p_1's new value (1). A new auxiliary node is placed in *auxList[1]* with a pointer to this final fused node, and *tos[1]* is updated to point to the final fused node.

In Figure 4, we illustrate the delete operation. In this case, we have deleted a node at p_2. At the primary, we delete the node and its pointer in the linked list. At the backup, this produces a "hole" in the fusion, since its first node represents no longer extant primary data. To address this and maintain space optimality, F removes the deleted data from the first node and updates it with the value of the node represented by *tos[2]*. The corresponding auxiliary node is also updated to point to the first node. We also remove

that value from *tos[2]* and move *tos[2]* to point to the next node down. In the end, the old data is no longer represented in any fused node and *auxLists[2]* represents the linked list at p_2. Note that if we were to perform *delete(8)* at p_1, the deletion process would result in a reduction in size of the fused linked list. In fact, the fused data structure is never larger than the largest primary data structure.

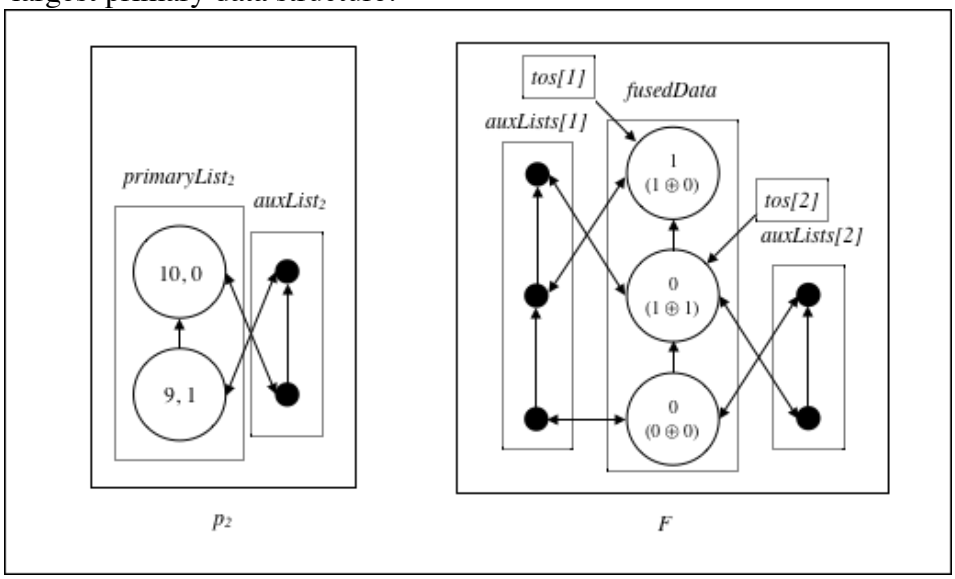

Figure 4: p_2 and F after *delete(5)*

As a result of fusing each primary node's data together, only f backups are required to correct f faults, and the fused data structure at each backup is only as large as the largest data structure at a node. However, because the data at the backup is stored in coded (or fused) form, recovery requires the participation of all other nodes. For a large n, this makes recovery quite expensive relative to active replication in terms of messages spent and computation. Recovery therefore can be expected to take much longer than is seen in active replication schemes.

Fusing other data structures

While here we are working with bits and the XOR operation, it's important to note that any object may be fused so long as a suitable fusion operator can be found. For example, if we represent strings as bit arrays, we can use a process similar to the preceding example to fuse those strings together. In [5], the authors describe a generic implementation of fusion covering the Java Collection Framework using Reed Solomon erasure codes as the fusion operator for generic data words, albeit with greater complexity in recovery time. Fusion can therefore be applied quite generically. However, other than the generic fusion implementation discussed in [5], few others exist to date, with the exception of the Python application under discussion in this report and a .NET implementation [12], both of which lack the breadth of coverage provided by [5]. Further research into fusion and its applications is available at [16, 17, 18, 19].

Chapter 3: Description of the test applications

The focus of this study is on a comparative analysis of active replication and fusion within a functioning distributed system. To that end, two applications have been built – one for each mechanism. Both applications are implemented in Python, although certain components – such as socket communication and serialization – utilize underlying C code.

At an abstract level, both applications have the task of maintaining arrays (*lists*, in Python) of bits. Without a fault tolerance mechanism, the bit lists are lost in the event of a crash fault. Clients in each application must therefore maintain backups of those lists using their respective mechanisms.

Each application supports at least two operations on the lists: *read* and *insert*. For architectural reasons discussed below, *reads* are locally available to the client in the fusion application and not in the active replication application. Read performance of the two implementations is therefore incommensurable and not evaluated here. Client *inserts* are acknowledged by the backup in both cases, so meaningful comparisons of update time can be made.

ACTIVE REPLICATION

The active replication mechanism contains two applications – a *group manager* and a *data node*. The data node application is only responsible for maintaining the actual replicated state (the list of bits), while the group manager application is responsible for maintaining all state pertaining to which data node instances are currently active and manages their entry/recovery process[1]. Any requests from clients – e.g., to update the

[1] This architecture comes from Attiya and Welch's helpful discussion of broadcast and multicast in [8].

data node or to read a particular value from the system's state – are routed through the group manager to the data nodes, as detailed below. Figure 5 illustrates the architecture of the active replication application with two groups.

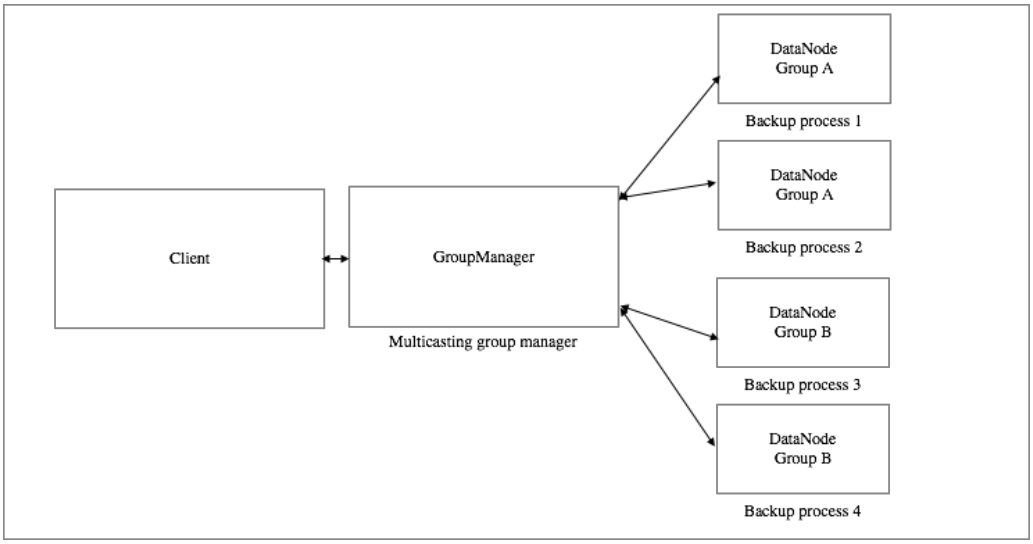

Figure 5: Architecture of experimental active replication application

All write operations are transmitted from the group manager to a group's data nodes using Skeen's algorithm, also known as ABCAST [3]. (Although more modern atomic multicast algorithms such as Zab [10] exist, Skeen's algorithm is optimal for number of messages sent [9] and is relatively simple to implement.) Read-only requests from a client are sent directly to an active node in the group, a slight loosening of the total-order constraint commonly accepted for performance.

When a data node enters or re-enters the system, all updating actions are halted while the node synchronizes with the group manager. This ensures that the recovering node is in a state consistent with its peers once it is returned to active participation in the group.

Network latency measurements are necessarily impacted by the set of messages passed between the client and the group manager. Since our fusion application has no such middleman, we eliminated this by taking all measurements at the group manager.

FUSION

The Python implementation of the fusion algorithm was first described in [7]. Briefly, it comprises a set of primary objects that maintain their own states as well as a server application that maintains the fused backup. The server application is instantiated with the class its nodes will be maintaining so that it can maintain the appropriate fused data structure. The primary objects implement a list-like interface, allowing them to be used by any other Python application.

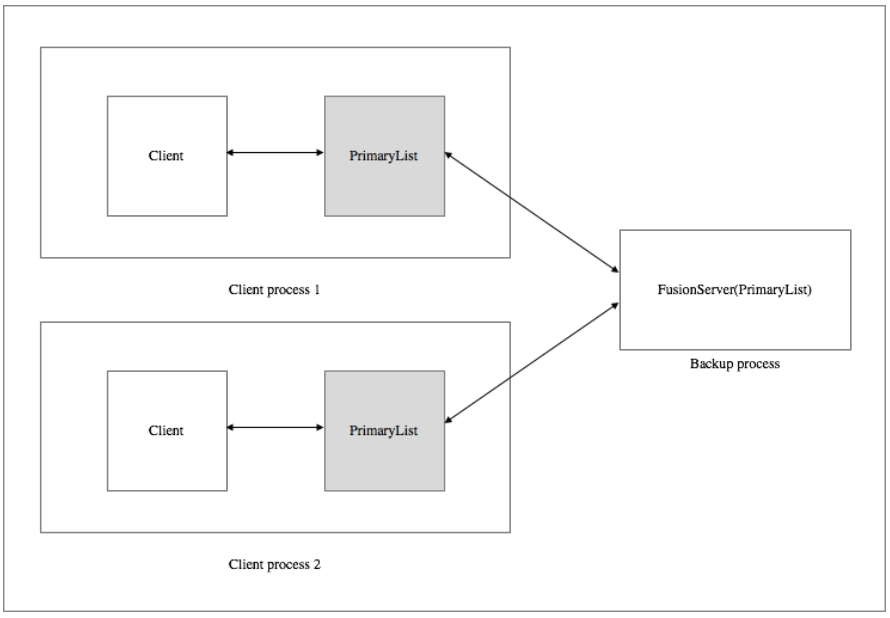

Figure 6: Architecture of experimental fusion application

In contrast with the active replication application, clients using the fusion system locally instantiate a primary object and maintain their state using that object. The primary

object transparently communicates all changes to the backup server and updates its own auxiliary data structures accordingly. As a result, the primary objects contain the state for their local node and can be read by the Python application without reference to the backup[2]. Figure 6 illustrates the fusion application.

In the event of a crash fault at a client, the primary object at the client calls the fusion server with its client's ID. The fusion server, in turn, polls all other participants in the fusion for their values and returns the client's last known state.

[2] This transparency has the benefit of allowing client application developers to use native-like data structures without any concern for managing replication. A similarly transparent client may be built for the active replication system using techniques similar to those used in the fusion system. However, this is left as an exercise; client usability and performance are out of scope for our purposes, and no measurements were taken at the actual clients of either application.

Chapter 4: Implementation and results

For our experiments, we deployed both applications under a number of configurations and evaluated their performance with respect to three properties: time required to update, space consumed at the backup, and time required for a crashed node to recover. Results were mixed with respect to update time. In most cases, experiments confirm the theoretical space savings of fusion. Active replication performed several orders of magnitude better than fusion with respect to recovery time.

EXPERIMENT SETUP

For each application, we executed several runs of the system with a fixed[3] fault tolerance f of 1 and varying numbers of "groups" g. The resulting node configurations are given in Table 1.

# of groups g	# of nodes n	
	Fusion	**Active replication**
1	1	2
2	2	4
3	3	6
4	4	8
5	5	10

Table 1: Node configurations for experimental runs ($f = 1$)

[3] Although the active replication system built for this paper is capable of handling multiple fault tolerance, the original fusion implementation we are comparing against is not. Our experiments therefore simulate single crash faults.

For each run, we executed 10,000 insert operations. After every 1,000 operations, every node was crashed and readmitted to the system to trigger the node recovery process. The node's primary data elements are therefore uniform in size, which allows us to maximize the potential space savings of fusion. All experiment runs were executed on a 2.3 GHz Intel i7 processor with 16GB of memory under Mac OS X 10.10.5. Throughout the experimental runs, we measured the following: the amount of time expended during backup, the amount of space consumed for backups, and the amount of time required to recover a node after a crash. All communication occurs over the socket layer; no additional network latency was introduced.

RESULTS

Backup time

Table 2 lists each backup mechanism's relative performance of backup time per group for each node configuration. In general, the time updating in fusion is robust against the number of nodes (n), with a mean time spent of approximately 0.001s seconds and negligible variance. However, as the number of groups increases, active replication suffers from occasional high-latency periods[4], which affect the mean group update time. The median update time is, however, on the same order as updates in fusion, indicating that the latency may be be an artifact of the implementation or experiment. Even so, the update time is unsurprising; since each update in the active replication application is multicast, the number of messages is necessarily higher.

[4] In our experiments, communication would occasionally take up to 2 seconds. The low frequency and distribution of these events impeded further investigation.

Backup time summary statistics										
groups	**1**		**2**		**3**		**4**		**5**	
	fusion	**AR**	**fusion**	**AR**	**fusion**	**AR**	**fusion**	**AR**	**fusion**	**AR**
mean (μs)	954.92	15232.07	1077.92	20005.41	1026.90	27031.80	1079.34	28579.39	1065.98	27406.47
median (μs)	940.00	3627.00	1023.75	3422.00	1001.67	3554.00	1021.50	3515.00	1068.00	3521.00
variance (μs2)	1.42	20847.06	1.14	30247.20	1.44	43816.43	1.62	46358.14	0.73	43566.28
std dev (μs)	1189.92	144385.13	1069.44	173917.22	1198.70	209323.75	1273.95	215309.40	855.87	208725.37

Table 2: Summary statistics for backup time per group

Space consumed at backup

As Figure 7 illustrates, the space required by the fusion backup is linear with respect to the number of elements in the backed-up data structure[5] and constant with respect to the number of nodes in the experimental configuration. This is not the case with the active replication backups. As the number of nodes n increases with the number of groups g, so too do the copies of our data, and the backup space necessarily increases along with n.

Interestingly, for small n (e.g., less than 10 here), the active replication scheme requires less space. This is a result of additional bookkeeping data required at the backup. Whereas the replicated nodes in the active replication application simply maintain a list of bits, at the backup we have to maintain an object with the fused value as well as pointers to the backup's auxiliary data structure as described in our discussion of fusion above. For comparison, the "Fusion (bits only)" series shows the size of the backup if we take into account only the fused bits. Since the additional data overhead imposed by these objects is constant, we see that as the number of groups increases, active replication quickly requires more backup space than does fusion.

[5] Although we measured the backup space required under each node configuration, it was – as expected – the same for each one.

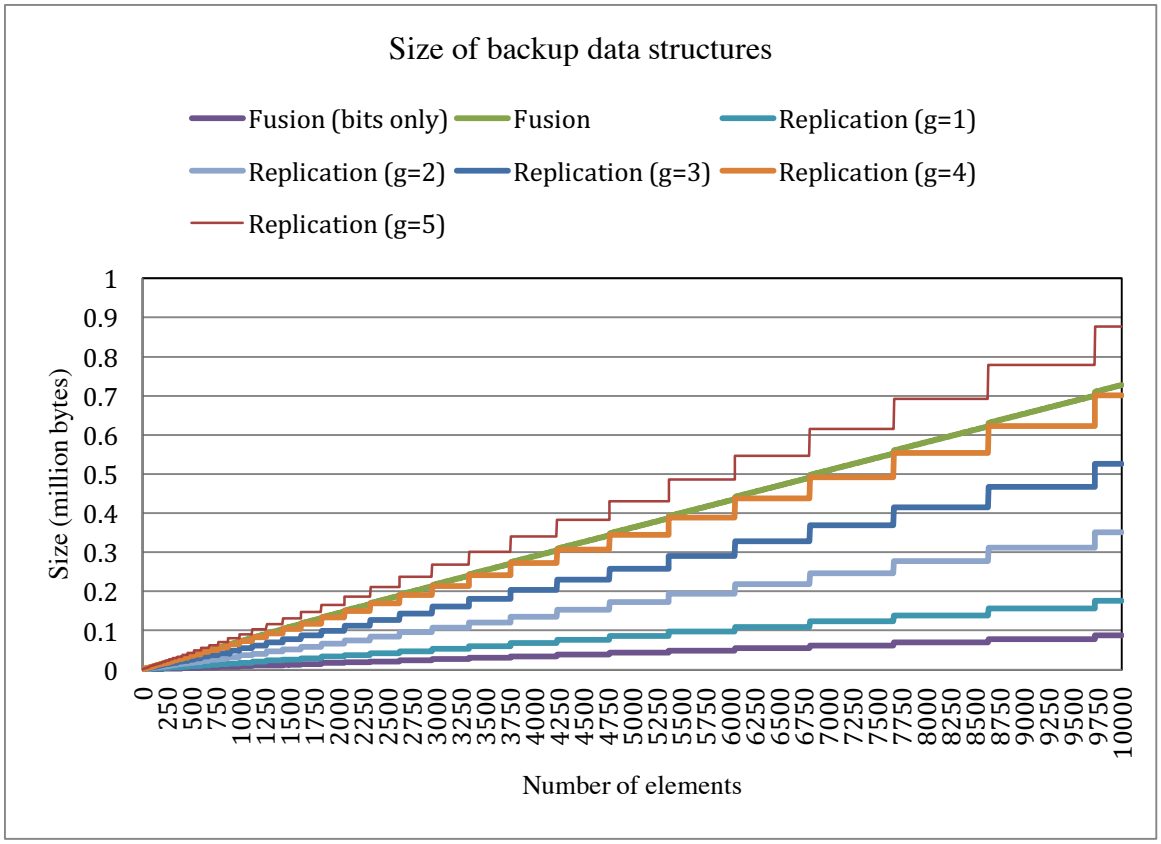

Figure 7: Backup space results

Recovery time

Figure 8 shows the amount of time spent recovering a node in our active replication system. Regardless of the node configuration, the time cost is relatively constant and very low.

It's worth noting that the recovery mechanism in this application relies on the fact that all updating operations are halted during its execution. Since no write operations are occurring, it simply reads the backed-up data from a single active node and passes that to the recovering node. More sophisticated strategies – such as those intended for high availability even during node recovery – might have additional requirements which

would increase the number of messages sent and therefore increase recovery time. Even under such a system, however, recovery time under active replication would be very fast.

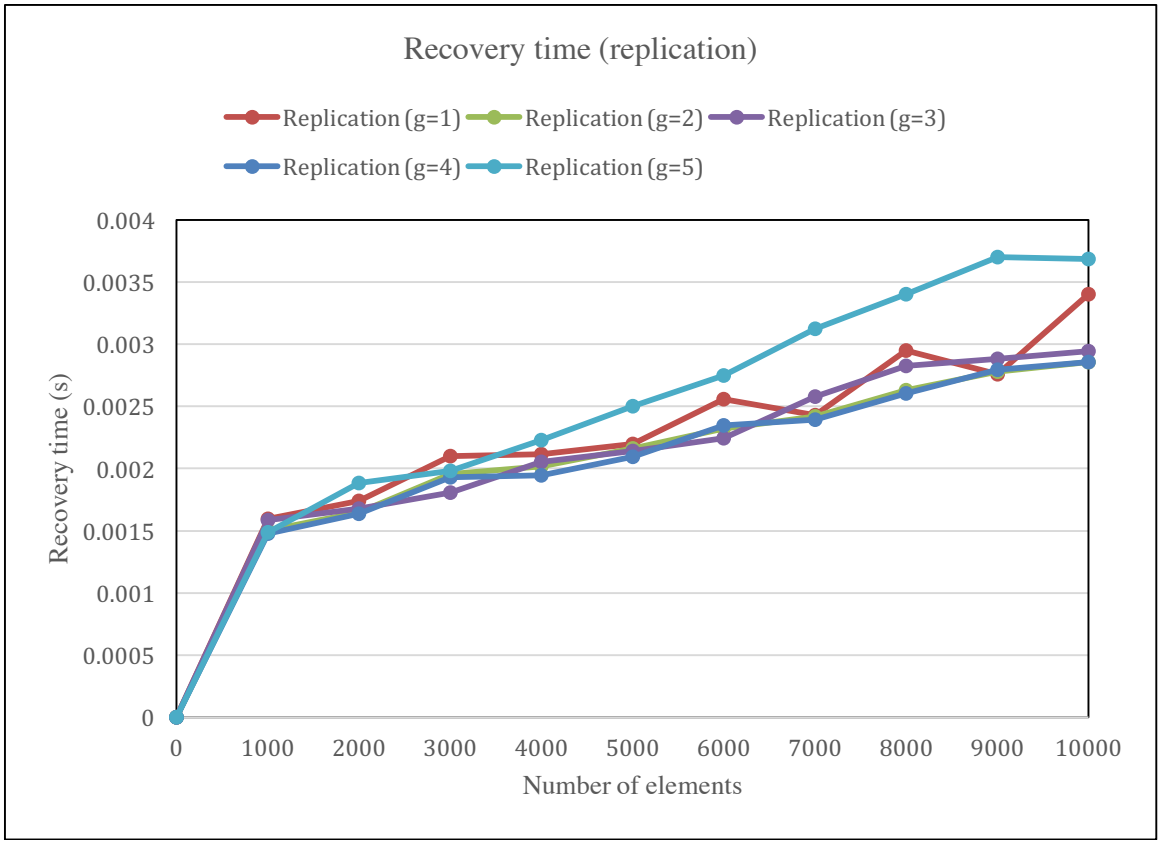

Figure 8: Recovery time results for replication

The results are very different for fusion. Figure 9 demonstrates the effect of the node configuration on recovery time under this scheme. Even in the the first non-trivial case (g=2) where two primaries are involved, the fusion application takes approximately 0.78 seconds to reconstruct and deliver a backed-up structure of 10000 elements[6].

[6] Notably, at the backup the entire 10000-element data structure, including the auxiliary data structure, consumes approximately 0.73 MB of space in memory.

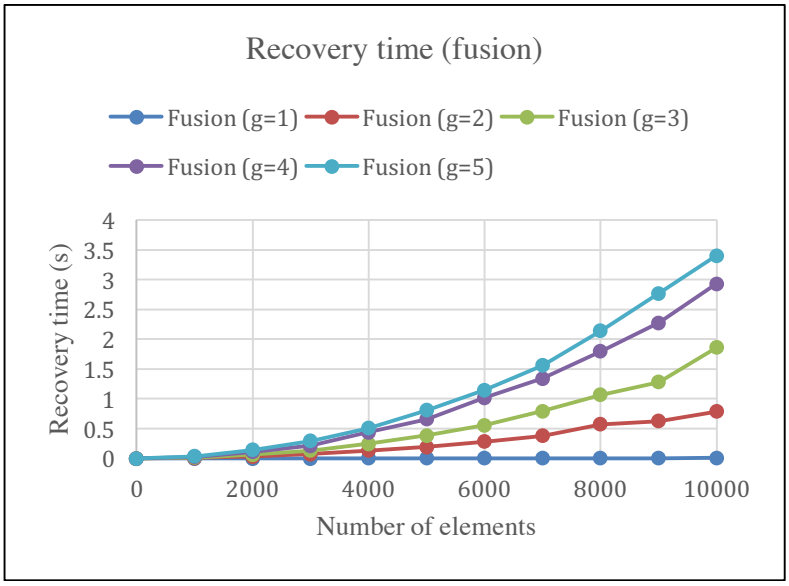

Figure 9: Recovery time results for fusion

Also worth noting is the fact that recovery time increases linearly with respect to the number of nodes for a given number of elements, as shown in Figure 10. This is in line with results in previous fusion implementations.

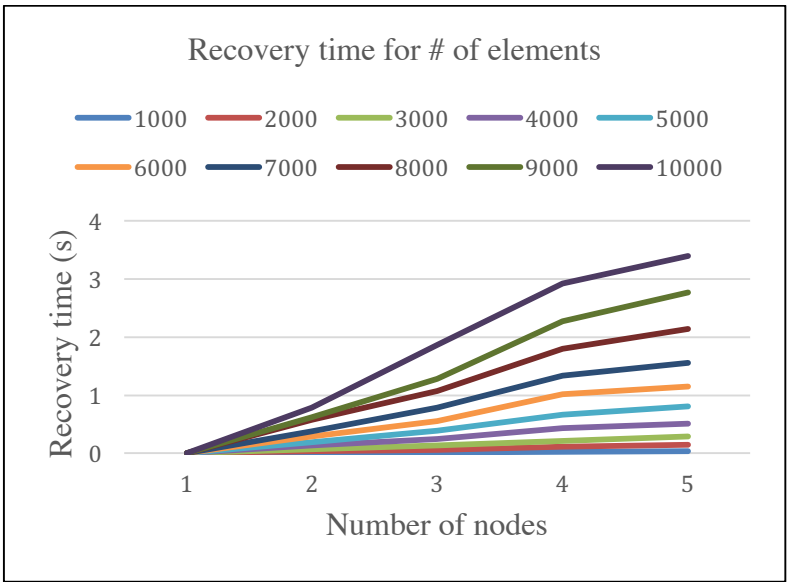

Figure 10: Recovery time for fusion by # of elements

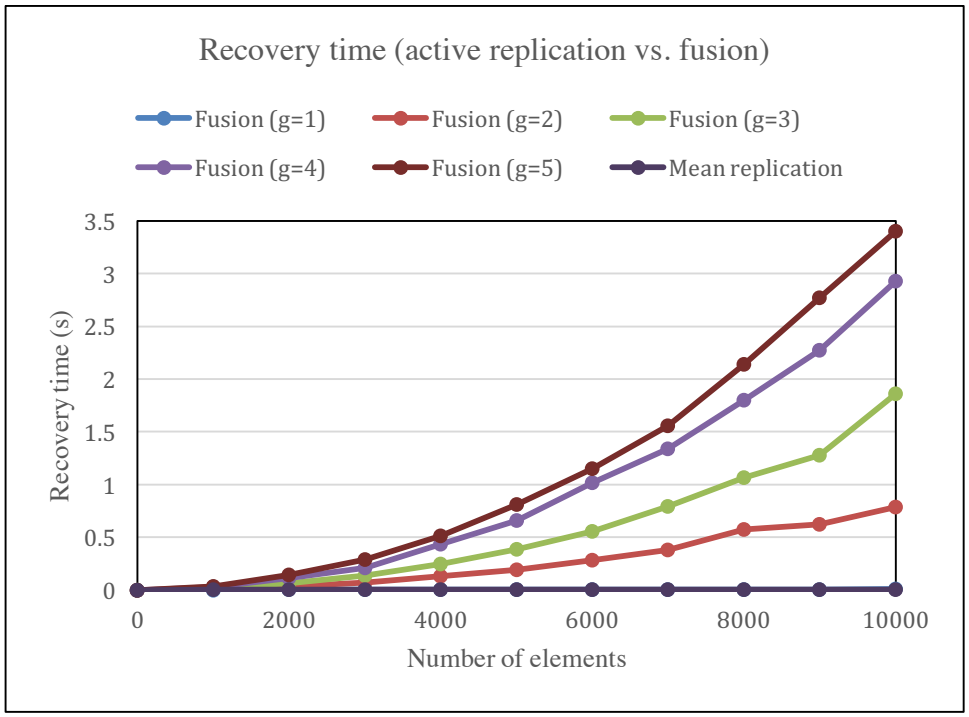

Figure 11: Side-by-side recovery time results (AR vs. fusion)

For comparison, Figures 11 and 12 show the recovery time results for each mechanism side-by-side at different scales. Of course, fusion is quickly outstripped by replication under any node configuration. In Figure 10, we show the mean replication time for all configurations because the fusion is several orders of magnitude more time-consuming.

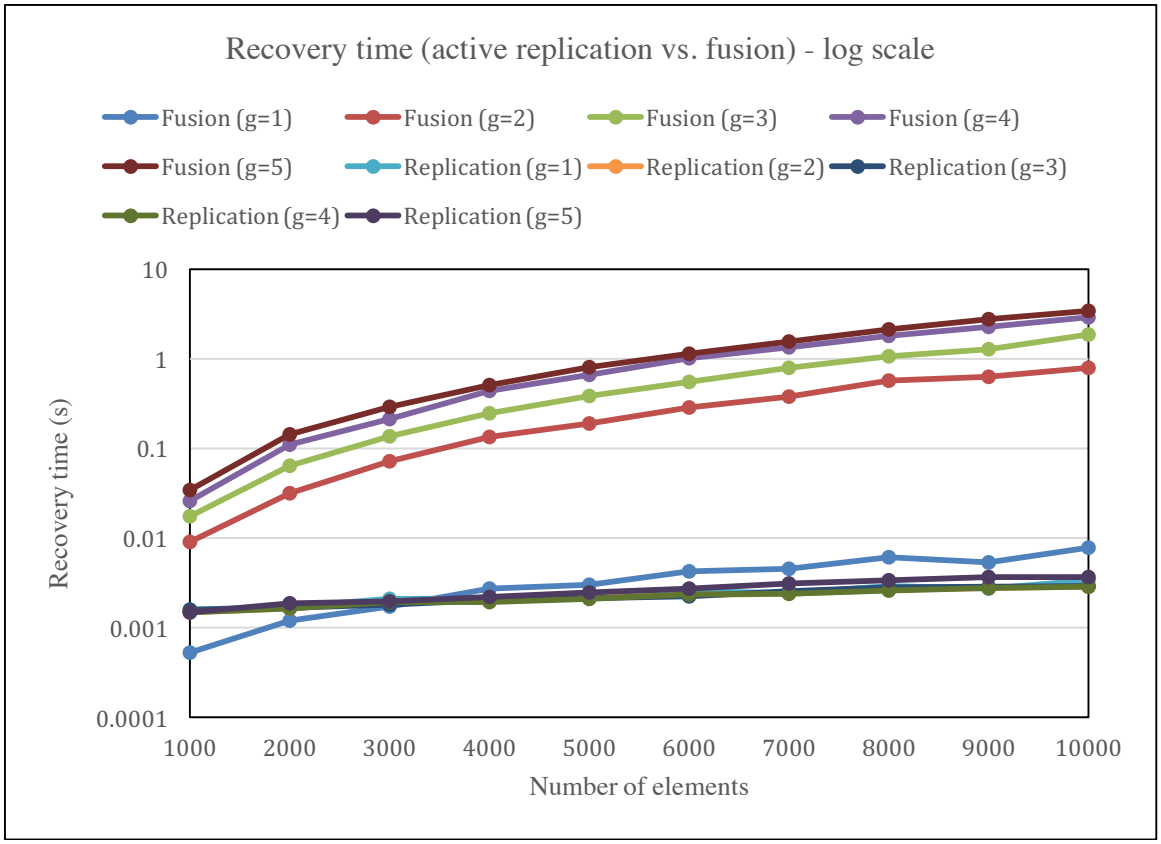

Figure 12: Side-by-side recovery time results (AR vs. fusion), log scale

EVALUATION

Our results regarding backup space required and recovery time comport well with the theoretical analysis of fusion and with previous implementations. They are therefore not surprising. Fusion, at its core, invites adopters to trade time spent recovering a data structure for space savings at the backup. And, as Figure 1 shows, those savings are not insignificant. Moreover, fusion shows significant speed advantages in backup time over active replication, since actions at the backup are on the same order as those at the backed-up node (see Theorem 2 at [5]).

Chapter 6: Conclusion

Conclusions

Our implementation has confirmed theoretical predictions and previous analysis of the relative performance of fusion and active replication. While space savings are significant for a system comprising a sufficiently large number of backed-up nodes, those savings come at the cost of very high recovery time. Adopters of fusion should carefully consider that cost. Fusion is likely to work best for systems where crash faults are uncommon or high availability is not required.

Suggestions for further work

As mentioned above, fusion may be best applied practically to systems with few crash faults or little to no high-availability requirements. Researchers should focus on describing exemplars of such systems and evaluating fusion's performance within them.

Moreover, to aid in increasing the adoption of fusion at large, more widely applicable frameworks need to be provided for the development community. The implementation at [5] covering the Java Collection Framework is a good start, but the technology is still ripe for further development in the form of generic fusion-based backup frameworks.

References

[1] L. Lamport, "The Implementation of Reliable Distributed Multiprocess Systems.," *CN*, vol. 2, no. 2, pp. 95–114, 1978.

[2] F. B. Schneider, "Implementing Fault-Tolerant Services Using the State Machine Approach: A Tutorial.," CSUR, vol. 22, no. 4, pp. 299–319, 1990.

[3] K. P. Birman and T. A. Joseph, "Reliable Communication in the Presence of Failures.," TOCS, vol. 5, no. 1, pp. 47–76, 1987.

[4] V. K. Garg and V. A. Ogale, "Fusible Data Structures for Fault-Tolerance.," *ICDCS*, 2007.

[5] B. Balasubramanian and V. K. Garg, "A Fusion-based Approach for Handling Multiple Faults in Distributed Systems," 2010.

[6] J. Vuckovic, "Modular Algorithms for Component Replication," 2006.

[7] J. Boyd and M. Linder, "Current Fusion-based Backup Approaches in Python." pp. 1–9, May-2013.

[8] H. Attiya and J. Welch, Distributed Computing. John Wiley & Sons, 2004.

[9] N. Schiper and F. Pedone, "On the Inherent Cost of Atomic Broadcast and Multicast in Wide Area Networks.," ICDCN, vol. 4904, no. 12, pp. 147–157, 2008.

[10] F. Junqueira and B. Reed, "A simple totally ordered broadcast protocol," presented at the Proceedings of the 2nd Workshop on Large-Scale Distributed Systems and Middleware, 2008.

[11] J. Boyd, "boydjj / fault-tolerance-eval." [Online]. Available: https://bitbucket.org/boydjj/fault-tolerance-eval.

[12] K. Meera and S. Abiramasundari, "Fault Tolerance in Parallel System Using Multiple Stacks," ijsr.net.

[13] A. S. Tanenbaum and M. van Steen, "Distributed systems - principles and paradigms (2. ed.).," Pearson Education 2007, 2007.

[14] Coulouris, J. Dollimore, and T. Kindberg, Distributed Systems: Concepts and Design (4th Edition) (International Computer Science). Addison-Wesley Longman Publishing Co., Inc., 2005.

[15] X. Défago, A. Schiper, and P. Urbán, "Total order broadcast and multicast algorithms: Taxonomy and survey.," CSUR, vol. 36, no. 4, pp. 372–421, 2004.

[16] V. K. Garg, "Implementing Fault-Tolerant Services Using State Machines: Beyond Replication.," DISC, vol. 6343, no. 44, pp. 450–464, 2010.

[17] B. Balasubramanian and V. K. Garg, “Fused Data Structures for Handling Multiple Faults in Distributed Systems.,” ICDCS, pp. 677–688, 2011.

[18] B. Balasubramanian and V. K. Garg, “Fault Tolerance in Distributed Systems Using Fused Data Structures.,” TPDS, vol. 24, no. 4, pp. 701–715, 2013.

[19] B. Balasubramanian and V. K. Garg, “Fault Tolerance in Distributed Systems using Fused State Machines,” Distributed Computing, vol. 27, no. 4, pp. 287–311, 2014.

[20] P. M. Chen, E. K. L. 0001, G. A. Gibson, R. H. Katz, and D. A. Patterson, “RAID: High-Performance, Reliable Secondary Storage,” CSUR, vol. 26, no. 2, pp. 145–185, 1994.

Appendix A: Source code for active replication application

The active replication application implemented for this report is a Python application, with the code available at [11]. Here, we list the layout of the application's directory. Following that, we list the source for each Python module. The reader is referred to [11] for greater detail, including comments and Python environment requirements. In particular, the files named "README.md" have detail on reproducing this report's experiments.

ACTIVE REPLICATION APPLICATION LAYOUT

```
fault-tolerance-eval/
    examples/
        node_config_f.ini
        test_correctness_fixed_f.py
    experiment_results/
    fault_tolerance_eval/
        data_node/
            __init__.py
            app.py
            replicated.py
        group_manager/
            __init__.py
            app.py
            proxy.py
            replicated.py
        networking/
            tests/
            __init__.py
            clock.py
            multicast.py
            protocol.py
            util.py
        __init__.py
    profiling/
        __init__.py
        fusion_comparison_fixed_f.py
        loggers.py
        size.py
```

Figure 13: Module layout for active replication application

Note that Python requires a file named __init__.py to identify a directory as a package. While initialization code is often present in these files, in our case they are all empty, so we have excluded them from the following listings.

Directory: examples

node_config_f.ini

```
[node 1]
multicast_host = localhost
multicast_port = 10000
read_host = localhost
read_port = 10001
group_id = A
proc_id = 1337
mgr_host = localhost
mgr_port = 9999

[node 2]
multicast_host = localhost
multicast_port = 10002
read_host = localhost
read_port = 10003
group_id = A
proc_id = 1338
mgr_host = localhost
mgr_port = 9999

[node 3]
multicast_host = localhost
multicast_port = 10004
read_host = localhost
read_port = 10005
group_id = B
proc_id = 1339
mgr_host = localhost
mgr_port = 9999

[node 4]
multicast_host = localhost
multicast_port = 10006
read_host = localhost
read_port = 10007
group_id = B
proc_id = 1340
mgr_host = localhost
mgr_port = 9999

[node 5]
multicast_host = localhost
multicast_port = 10008
read_host = localhost
read_port = 10009
group_id = C
proc_id = 1341
mgr_host = localhost
mgr_port = 9999

[node 6]
```

```
multicast_host = localhost
multicast_port = 10010
read_host = localhost
read_port = 10011
group_id = C
proc_id = 1342
mgr_host = localhost
mgr_port = 9999

[node 7]
multicast_host = localhost
multicast_port = 10012
read_host = localhost
read_port = 10013
group_id = D
proc_id = 1343
mgr_host = localhost
mgr_port = 9999

[node 8]
multicast_host = localhost
multicast_port = 10014
read_host = localhost
read_port = 10015
group_id = D
proc_id = 1344
mgr_host = localhost
mgr_port = 9999

[node 9]
multicast_host = localhost
multicast_port = 10016
read_host = localhost
read_port = 10017
group_id = E
proc_id = 1345
mgr_host = localhost
mgr_port = 9999

[node 10]
multicast_host = localhost
multicast_port = 10018
read_host = localhost
read_port = 10019
group_id = E
proc_id = 1346
mgr_host = localhost
mgr_port = 9999
```

test_correctness_fixed_f.py

```
"""
Make sure active replication app works for a given number of groups. Example usage:

python examples/test_correctness_fixed_f.py 2
"""

import json
import logging
import random
import socket
import sys

logging.basicConfig(level=logging.DEBUG)
logger = logging.getLogger(__name__)

MGR_ADDR = ('localhost', 9999)
```

```
GROUPS = {
    1: 'A',
    2: 'B',
    3: 'C',
    4: 'D',
    5: 'E',
}


def test_correctness(num_messages, num_groups):
    """
    Send num_messages messages to num_groups groups, as defined by `GROUPS` above.
    """

    # For every group... [note: xrange(start, end) is exclusive of end]
    for i in xrange(1, num_groups + 1):
        group_id = GROUPS[i]
        logger.debug('Sending messages to group %s', group_id)
        # ... send num_messages messages.
        for j in xrange(num_messages):
            msg = {'client_type': 'external', 'group_id': group_id, 'method': 'add int',
                   'value': random.randint(0, 1)}
            serialized_msg = json.dumps(msg)

            sock = socket.create_connection(MGR_ADDR)
            sock.sendall(serialized_msg + '\n')
            resp = sock.recv(2048)
            logger.info(resp)

if __name__ == '__main__':
    num_messages = int(sys.argv[1])
    num_groups = int(sys.argv[2])
    logger.info('Got called with %s, %s', num_messages, num_groups)
    test_correctness(num_messages, num_groups)
```

Directory: fault_tolerance_eval/data_node

app.py

```
import argparse
from ConfigParser import SafeConfigParser
import contextlib
import json
import logging
import socket
import SocketServer
import threading
import time

from .replicated import ReplicatedDataNode

from ..networking import multicast
from ..networking import util
from ..networking.protocol import Protocol

logger = logging.getLogger(__name__)


class GroupManagerException(Exception):
    pass


class MulticastMessageHandler(object):
    def __init__(self, node):
        self._node = node
```

```
    def handle(self, msg):
        logger.debug('Got msg: %s', msg)
        deserialized = json.loads(msg.content)
        logger.debug(deserialized)
        logger.debug('Node data before: %s', self._node.data)

        if deserialized['method'] == Protocol.ADD_INT:
            self._node.add_int(deserialized['value'])
        logger.debug('Node data after: %s', self._node.data)


class DataNodeReadServer(util.ReusableThreadingTCPServer):
    def __init__(self, server_address, handler_class, data_node, bind_and_activate=True):
        util.ReusableThreadingTCPServer.__init__(self, server_address, handler_class,
                                                 bind_and_activate)

        self.data_node = data_node
        logger.debug('Initialized read server with data node %s', self.data_node)


class DataNodeReadRequestHandler(SocketServer.StreamRequestHandler):
    def handle(self):
        payload = self.rfile.readline().strip()
        logger.debug('Got payload from client: %s', payload)
        request = json.loads(payload)

        response = {}
        if request['method'] == Protocol.READ_ALL:
            node_data = self.server.data_node.data
            response = Protocol.get_read_all_response(self.server.data_node.group_id,
                                                      node_data)

        self.write_response(Protocol.serialize(response))

    def write_response(self, content):
        self.wfile.write('{response}'.format(response=content))


class ReplicatedDataNodeApp(object):
    NUM_MULTICAST_WORKERS = 2

    def __init__(self, config_file, section_id):
        logger.debug('Initializing with config_file %s, section_id %s', config_file,
                     section_id)
        parser = SafeConfigParser()
        parser.read(config_file)
        logger.debug('Config parser: %s', parser)

        section = 'node {}'.format(section_id)
        self._group_id = parser.get(section, 'group_id')

        mgr_host = parser.get(section, 'mgr_host')
        mgr_port = parser.getint(section, 'mgr_port')
        self._mgr_addr = (mgr_host, mgr_port)

        multicast_host = parser.get(section, 'multicast_host')
        multicast_port = parser.getint(section, 'multicast_port')
        self._multicast_addr = (multicast_host, multicast_port)

        read_host = parser.get(section, 'read_host')
        read_port = parser.getint(section, 'read_port')
        self._read_addr = (read_host, read_port)

        self._proc_id = parser.getint(section, 'proc_id')
        self._group_id = parser.get(section, 'group_id')

    def __get_mgr_client(self):
```

```
        logger.debug('Creating socket to communicate with %s', self._mgr_addr)
        return socket.create_connection(self._mgr_addr)

    def run(self):
        """
        The entry point for the DataNode application to run.
        """
        # Check with manager to see if we need to catch up on any existing data.
        initial_data = self._sync_with_mgr() or []

        self._node = ReplicatedDataNode(self._group_id, self._proc_id, initial_data)

        # Start listening for multicast messages.
        handler = MulticastMessageHandler(self._node)
        multicast_server = multicast.multicast_serve(self._multicast_addr,
                                                     self.NUM_MULTICAST_WORKERS,
                                                     handler.handle)

        logger.debug('Initializing read server at %s', self._read_addr)
        read_server = DataNodeReadServer(self._read_addr, DataNodeReadRequestHandler,
self._node)
        server_thread = threading.Thread(target=read_server.serve_forever)
        server_thread.start()

        # Let the group manager know we're open for business.
        self._finalize_entry()

        return multicast_server, read_server

    def _sync_with_mgr(self):
        with contextlib.closing(self.__get_mgr_client()) as mgr_client:
            logger.debug('Initializing sync with manager at %s.', self._mgr_addr)
            msg = Protocol.get_entry(self._group_id, self._proc_id)
            payload = Protocol.serialize(msg)
            mgr_client.sendall(payload)
            mgr_response = json.loads(util.recv_until(mgr_client))
            logger.debug('Manager response (type %s): %s', type(mgr_response),
mgr_response)

            if mgr_response['method'] == Protocol.ENTRY_SYNC:
                logger.debug('Got initial data back from manager.')
                response = mgr_response['data']
            else:
                logger.debug('No initial data received from manager.')
                response = None
            return response

    def _finalize_entry(self):
        with contextlib.closing(self.__get_mgr_client()) as mgr_client:
            logger.debug('Finalizing entry with manager.')
            msg = Protocol.get_entry_finalization(self._group_id, self._proc_id)
            payload = Protocol.serialize(msg)
            mgr_client.sendall(payload)
            response = json.loads(util.recv_until(mgr_client))

            if response['method'] != Protocol.ENTRY_FINALIZE_ACK:
                raise GroupManagerException(
                    'The manager did not accept our finalization.')

            logger.info('The manager ACKed our finalization. We can go live.')


if __name__ == '__main__':
    logging.basicConfig(level=logging.DEBUG,
                        format='%(asctime)s:%(levelname)-5s(%(threadName)-10s:%(name)s):
%(message)s')
    arg_parser = argparse.ArgumentParser(description='ReplicatedDataNodeApp')
    arg_parser.add_argument('node_config', action='store',
```

```
                            help='The location of a config file for this client to use.')
    arg_parser.add_argument('config_section', action='store',
                            help='The section # within the node config for this client.')

    args = arg_parser.parse_args()

    app = ReplicatedDataNodeApp(args.node_config, args.config_section)
    logger.info('Starting server...')
    multicast_server, read_server = app.run()

    try:
        logger.info('Beginning wait loop. Hit Ctrl-C to kill server.')
        while True:
            time.sleep(10)
    except KeyboardInterrupt:
        logger.debug('Got KeyboardInterrupt, shutting down multicast server...')
        multicast_server.shutdown()
        logger.debug('Shutting down read server...')
        read_server.shutdown()
        logger.debug('Closing multicast server...')
        multicast_server.server_close()
        logger.debug('Closing read server...')
        read_server.server_close()
    finally:
        main_thread = threading.current_thread()
        child_threads_done = True
        for t in threading.enumerate():
            if t is main_thread:
                continue
            thread_name = t.getName()
            logger.info('Giving %s 1s to shut down...', thread_name)
            t.join(1)
            if t.isAlive():
                logger.info('%s did not shut down in time, killing.', thread_name)

    logger.debug('Shutdown complete.')
```

replicated.py

```
import logging

from profiling.loggers import get_backup_size_logger_for_node
from profiling.size import total_size

logger = logging.getLogger(__name__)

backup_size_logger = None


class ReplicatedDataNode(object):
    def __init__(self, group_id, proc_id, data=None):
        self.group_id = group_id
        # REALLY this is the Node ID
        self.node_id = proc_id

        if data is None:
            self.data = []
        else:
            self.data = data

        global backup_size_logger
        if backup_size_logger is None:
            backup_size_logger = get_backup_size_logger_for_node(self.group_id,
self.node_id)

    def __str__(self):
        return 'DataNode {}:{} with {} elements'.format(self.group_id, self.node_id,
```

```
                                                    len(self.data))

    def add_int(self, value):
        self.data.append(value)
        backup_size_logger.info('%s,%s,%s,%s', self.group_id, self.node_id,
len(self.data), total_size(self.data))
```

Directory: fault_tolerance_eval/group_manager

app.py

```
import argparse
import json
import logging
import SocketServer
import threading
import time

from ..networking import util
from ..networking import protocol

from .replicated import ReplicatedGroupManager

logger = logging.getLogger(__name__)


class ReplicatedGroupManagerServer(util.ReusableThreadingTCPServer):
    def __init__(self, server_address, handler_class, manager,
                 bind_and_activate=True):
        util.ReusableThreadingTCPServer.__init__(self, server_address,
                                                 handler_class, bind_and_activate)
        self.manager = manager

        # Handler threads must freeze all non-read operations when a node is (re-
)entering
        # the group. When a freeze-inducing method is called, we clear() this Event and
        # set() it again once the node's synchronization is complete.
        self.mutating_operations_allowed = threading.Event()
        self.mutating_operations_allowed.set()
        logger.debug('Server initialized to serve %s nodes.', self.manager.num_nodes)


class ReplicatedGroupManagerRequestHandler(SocketServer.StreamRequestHandler):
    """
    Deserialize requests and hand them off to `ReplicatedGroupManager`.
    Serialize responses on the way out. All responses are the responsibility of the
    `ReplicatedGroupManager`.
    """
    def handle(self):
        __handlers = {
            protocol.Protocol.EXTERNAL: self._handle_external_request,
            protocol.Protocol.NODE: self._handle_node_request,
        }

        payload = self.rfile.readline().strip()
        logger.debug('Got payload %s', payload)
        request = json.loads(payload)

        method = __handlers.get(request['client_type'], self._handle_unexpected_request)

        self.write_response(protocol.Protocol.serialize(method(request)))

    def finish(self):
        SocketServer.StreamRequestHandler.finish(self)
        try:
            if self.set_mutating_operations_event:
```

```
                logger.debug('Handler has released the mutating operations flag.')
                self.server.mutating_operations_allowed.set()
        except AttributeError:
            pass

    def _handle_external_request(self, request):
        """
        Handle requests coming from external (non-node) clients.
        """
        logger.debug('Got external request %s', request)
        response = None

        if request['method'] == protocol.Protocol.READ_ALL:
            response = self.server.manager.readall(request['group_id'])
        elif request['method'] == protocol.Protocol.ADD_INT:
            response = self.server.manager.add_int(request['group_id'], request['value'])

        return response

    def _handle_node_request(self, request):
        """
        Handle requests coming from node clients. E.g., entry, crash, etc.
        """
        logger.debug('Got node request %s', request)
        response = None

        if request['method'] == protocol.Protocol.ENTER:
            logger.debug('Got entry request from process %s', request['proc_id'])
            # Wait up to 10s for any other blocking operations to complete.
            can_proceed = self.server.mutating_operations_allowed.wait(10)
            if not can_proceed:
                logger.warn('Previous node management op failed. Proceeding...')
            self.server.mutating_operations_allowed.clear()
            response = self.server.manager.handle_entry(request['group_id'],
                                                        request['proc_id'])
        elif request['method'] == protocol.Protocol.ENTRY_FINALIZE:
            response = self.server.manager.handle_entry_finalization(request['group_id'],
                                                                     request['proc_id'])
            self.set_mutating_operations_event = True
        return response

    def _handle_unexpected_request(self, request):
        logger.debug('Got unexpected request %s', request)
        response = {'method': protocol.Protocol.UNKNOWN_CLIENT}
        return response

    def write_response(self, content):
        self.wfile.write('{response}'.format(response=content))


if __name__ == '__main__':
    logging.basicConfig(level=logging.DEBUG,
                        format='%(asctime)s:%(levelname)-5s(%(threadName)-10s:%(name)s):
%(message)s')
    arg_parser = argparse.ArgumentParser(description='ReplicatedGroupManagerApp')
    arg_parser.add_argument('host', action='store')
    arg_parser.add_argument('port', action='store', type=int)
    arg_parser.add_argument('num_nodes', action='store', type=int)
    arg_parser.add_argument('node_config', action='store')
    args = arg_parser.parse_args()

    group_manager = ReplicatedGroupManager(args.num_nodes, args.node_config)

    server = ReplicatedGroupManagerServer((args.host, args.port),
                                          ReplicatedGroupManagerRequestHandler,
                                          group_manager)

    server_thread = threading.Thread(target=server.serve_forever)
```

```
    logger.debug('Starting server...')
    server_thread.start()

    try:
        logger.info('Beginning wait loop. Hit Ctrl-C to kill server.')
        while True:
            time.sleep(10)
    except KeyboardInterrupt:
        logger.debug('Got KeyboardInterrupt, shutting down...')
        server.shutdown()
        logger.debug('Closing server...')
        server.server_close()
    finally:
        logger.debug('Waiting for server thread to finish.')
        server_thread.join()
    logger.debug('Shutdown complete.')
```

proxy.py

```
import contextlib
import json
import logging
import socket

from ..networking import protocol
from ..networking import util

logger = logging.getLogger(__name__)


class NodeProxy(object):
    """
    Handle network communication to a node. Methods should return deserialized objects.
    """
    def __init__(self, multicast_addr, read_addr, group_id, proc_id, active=False):
        self.multicast_addr = multicast_addr
        self.read_addr = read_addr
        self.group_id = group_id
        self.proc_id = proc_id
        self.active = active

    def __get_socket(self):
        return socket.create_connection(self.read_addr)

    def read(self, index):
        """
        Perform network communication to get value at an index.
        """
        msg = protocol.Protocol.get_read(index)
        logger.debug('Request: %s', msg)
        with contextlib.closing(self.__get_socket()) as sock:
            logger.debug('Sending request to node...')
            sock.sendall(msg)
            logger.debug('Waiting for response...')
            response = util.recv_until(sock)

        logger.debug('Response: %s', response)
        return json.loads(response)

    def readall(self):
        """
        Perform network communication to get entire data structure.
        """
        msg = protocol.Protocol.get_readall(self.group_id)
        logger.debug('Request: %s', msg)
        with contextlib.closing(self.__get_socket()) as sock:
            logger.debug('Sending readall request to node...')
```

```
            msg = protocol.Protocol.serialize(msg)
            sock.sendall(msg)
            logger.debug('Waiting for response...')
            response = util.recv_until(sock)

        logger.debug('Response: %s', response)
        return json.loads(response)

    def __unicode__(self):
        return u'Node {proc_id} (group {group_id}) at {addr}'.format(
            proc_id=self.proc_id,
            group_id=self.group_id,
            addr=self.read_addr
        )

    def __str__(self):
        return unicode(self)

    def __repr__(self):
        return u'{class_name}({multicast_addr}, {read_addr}, {group_id}, {proc_id}, ' \
               u'{active})'.format(
            class_name=self.__class__.__name__,
            multicast_addr=repr(self.multicast_addr),
            read_addr=repr(self.read_addr),
            group_id=repr(self.group_id),
            proc_id=repr(self.proc_id),
            active=repr(self.active),
        )
```

replicated.py

```
import collections
import ConfigParser
import datetime
import logging

from profiling.loggers import update_time_logger, recovery_time_logger
from ..networking import multicast
from ..networking import protocol

from . import proxy

logger = logging.getLogger(__name__)


class ReplicatedGroupManager(object):
    def __init__(self, num_nodes, node_config):
        self.num_nodes = num_nodes

        config_parser = ConfigParser.SafeConfigParser()
        config_parser.read(node_config)
        self.nodes = collections.defaultdict(dict)

        self.active_nodes = set()
        self.inactive_nodes = set()
        self.transitioning_nodes = set()

        node_sections = ['node {}'.format(i) for i in xrange(1, self.num_nodes+1)]
        for section in node_sections:
            multicast_host = config_parser.get(section, 'multicast_host')
            multicast_port = config_parser.getint(section, 'multicast_port')
            read_host = config_parser.get(section, 'read_host')
            read_port = config_parser.getint(section, 'read_port')

            node = proxy.NodeProxy(
                (multicast_host, multicast_port),
                (read_host, read_port),
```

```
                config_parser.get(section, 'group_id'),
                config_parser.getint(section, 'proc_id'),
            )
            self.nodes[node.group_id][node.proc_id] = node

            # All nodes are inactive to start.
            self.inactive_nodes.add(node)

        self.multicast_clients = {}
        for group_id, group in self.nodes.iteritems():
            _multicast_addrs = [node.multicast_addr for node in group.values()]
            self.multicast_clients[group_id] =
multicast.MulticastClient(_multicast_addrs)

        logger.debug('Nodes initialized to: %s', self.nodes)

    def __transition_node(self, node):
        self.multicast_clients[node.group_id].remove_host(node.multicast_addr)
        if node in self.active_nodes:
            self.active_nodes.remove(node)
        elif node in self.inactive_nodes:
            self.inactive_nodes.remove(node)

        node.active = False
        self.transitioning_nodes.add(node)


    def __activate_node(self, node):
        self.multicast_clients[node.group_id].add_host(node.multicast_addr)
        if node in self.transitioning_nodes:
            self.transitioning_nodes.remove(node)
        elif node in self.inactive_nodes:
            self.inactive_nodes.remove(node)

        node.active = True
        self.active_nodes.add(node)

    def __deactivate_node(self, node):
        self.multicast_clients[node.group_id].remove_host(node.multicast_addr)
        if node in self.transitioning_nodes:
            self.transitioning_nodes.remove(node)
        elif node in self.active_nodes:
            self.active_nodes.remove(node)

        node.active = False
        self.inactive_nodes.add(node)

    def read(self, group_id, index):
        """
        Perform a consistent (although not nec. up-to-date) read operation for an index
of
        the data structure.
        """
        response = protocol.Protocol.get_read_response(group_id, index, [])
        group_nodes = self.nodes[group_id]
        if not group_nodes:
            logger.debug('No nodes found for group %s. Possible groups are: %s',
                         group_id, self.nodes.keys())
            return protocol.Protocol.get_read_response(group_id, index, [])

        logger.debug('Nodes: %s', group_nodes)
        for node_id, node_proxy in group_nodes.iteritems():
            if node_proxy not in self.active_nodes:
                continue

            # noinspection PyBroadException
            try:
                logger.debug('Attempting read from %s', node_proxy)
```

```
                node_data = node_proxy.read(index)
                response = protocol.Protocol.get_read_response(group_id, index,
                                                               node_data)
            except:
                logger.exception('Could not read from node %s', node_id)
                continue
            else:
                break

        return response

    def readall(self, group_id):
        """
        Perform a consistent (although not nec. up-to-date) read-all operation for a
group.
        """
        response = protocol.Protocol.get_read_all_response(group_id, [])
        group_nodes = self.nodes[group_id]
        if not group_nodes:
            logger.debug('No nodes found for group %s. Possible groups are: %s',
                         group_id, self.nodes.keys())
            return protocol.Protocol.get_read_all_response(group_id, [])

        logger.debug('Nodes: %s', group_nodes)
        for node_id, node_proxy in group_nodes.iteritems():
            if node_proxy not in self.active_nodes:
                continue

            # noinspection PyBroadException
            try:
                logger.debug('Attempting readall from %s', node_proxy)
                node_data = node_proxy.readall()
                response = protocol.Protocol.get_read_all_response(group_id, node_data)
            except:
                logger.exception('Could not readall from node %s', node_id)
                continue
            else:
                break

        return response

    def add_int(self, group_id, value):
        message = protocol.Protocol.get_add_int(group_id, value)
        group_nodes = self.nodes[group_id]
        if not group_nodes:
            logger.debug('No nodes found for group %s. Possible groups are: %s',
                         group_id, self.nodes.keys())
            return protocol.Protocol.get_read_all_response(group_id, [])

        logger.debug('Nodes: %s', group_nodes)

        multicast_client = self.multicast_clients[group_id]

        # Capture before/after time so that we can log the delta
        t0 = datetime.datetime.now()
        multicast_client.send(protocol.Protocol.serialize(message))
        t1 = datetime.datetime.now()
        delta = t1 - t0
        update_time_logger.info('%s,%s', len(group_nodes), delta.total_seconds())

        return True

    def handle_entry(self, group_id, proc_id):
        response = None
        logger.debug('Nodes: %s', self.nodes)
        t0 = datetime.datetime.now()
        node = self.nodes[group_id][proc_id]
        self.__transition_node(node)
```

```
        active_group_nodes = set(self.nodes[group_id].values()) - \
                             self.transitioning_nodes - \
                             self.inactive_nodes
        logger.info('Active group nodes: %s', active_group_nodes)
        if not active_group_nodes:
            response = protocol.Protocol.get_entry_accepted(group_id, proc_id)

        for node in active_group_nodes:
            try:
                logger.debug('Attempting to readall from %s', node)
                node_response = node.readall()
                response = protocol.Protocol.get_entry_sync(group_id, proc_id,
                                                            node_response['data'])
            except:
                logger.exception('Could not read from node %s', node)
                continue
            else:
                logger.debug('Got more data for new node. Sending back %s',
                             response['data'])
                if node_response['data']:
                    t1 = datetime.datetime.now()
                    delta = t1 - t0
                    recovery_time_logger.info('%s,%s,%s',
                                              len(self.nodes[group_id]),
                                              len(node_response['data']),
                                              delta.total_seconds())
                break
        return response

    def handle_entry_finalization(self, group_id, proc_id):
        node = self.nodes[group_id][proc_id]
        self.__activate_node(node)
        return protocol.Protocol.get_ack_entry_finalization(group_id, proc_id)
```

Directory: fault_tolerance_eval/networking

clock.py

```
import logging
import threading

logger = logging.getLogger(__name__)


class Clock(object):
    """
    A thread-safe singleton that manages the worker's Lamport clock. See
    https://en.wikipedia.org/wiki/Lamport_timestamps and
    http://python-3-patterns-idioms-test.readthedocs.org/en/latest/Singleton.html

    Thread safety provided by double-checked locking on instantiation. See:
    https://gist.github.com/werediver/4396488 and
    https://en.wikipedia.org/wiki/Double-checked_locking


    All write operations protected with locks.
    """
    # noinspection PyPep8Naming
    class __Clock(object):
        def __init__(self):
            self.counter = 0
            self.lock = threading.Lock()

        def __str__(self):
            return 'Lamport clock: {}'.format(self.counter)
```

```
    instance = None
    singleton_lock = threading.Lock()

    def __init__(self):
        if not Clock.instance:
            with Clock.singleton_lock:
                if not Clock.instance:
                    Clock.instance = Clock.__Clock()

    def __getattr__(self, name):
        return getattr(self.instance, name)

    def __str__(self):
        return str(self.instance)

    def tick(self):
        """
        Use when internal events occur.
        """
        with self.instance.lock:
            self.instance.counter += 1
            return self.instance.counter

    def send_msg(self):
        """
        Use when sending a message; include resulting value in outgoing message
        """
        return self.tick()

    def recv_msg(self, other_counter):
        with self.instance.lock:
            self.instance.counter = max(self.instance.counter, other_counter) + 1
            return self.instance.counter

clock = Clock()
```

multicast.py

```
"""
Primitives for multicasting with the ABCAST protocol.
"""
import json
import logging
import Queue
import SocketServer
import socket
import threading
import uuid

from profiling.loggers import multicast_msg_count_logger
from .clock import clock
from . import util

logging.basicConfig(level=logging.INFO,
                    format='%(asctime)s (%(threadName)-10s) %(message)s')
logger = logging.getLogger(__name__)

MULTICAST_STATUS_TENTATIVE = 'tentative'
MULTICAST_STATUS_FINAL = 'final'


class NotDeliverable(Exception):
    pass


class MulticastMessage(object):
    MESSAGE_TEMPLATE = "{timestamp} {status} {msg_id} {content}"
```

```
    def __init__(self, content, timestamp=None, status=None, msg_id=None):
        self.timestamp = timestamp if timestamp else clock.counter
        self.status = status if status else MULTICAST_STATUS_TENTATIVE
        self.msg_id = str(msg_id) if msg_id else str(uuid.uuid4())
        self.content = content

        # All messages are initially undeliverable and may only be marked deliverable
        # after they are finalized in the second round of multicasting.
        self.deliverable = False

    def to_json(self):
        return json.dumps({
            'timestamp': self.timestamp,
            'status': self.status,
            'msg_id': self.msg_id,
            'content': self.content,
        })

    def __eq__(self, other):
        if not isinstance(other, MulticastMessage):
            return False
        return self.msg_id == other.msg_id

    def __ne__(self, other):
        if not isinstance(other, MulticastMessage):
            return True
        return self.msg_id != other.msg_id

    def __cmp__(self, other):
        return cmp(self.timestamp, other.timestamp)

    def __str__(self):
        return self.MESSAGE_TEMPLATE.format(
            timestamp=self.timestamp,
            status=self.status,
            msg_id=self.msg_id,
            content=self.content
        )

    def __repr__(self):
        return "{}(timestamp={}, status='{}', msg_id='{}', content='{}')".format(
            self.__class__.__name__,
            self.timestamp,
            self.status,
            self.msg_id,
            self.content
        )


class MulticastMessageQueue(object):
    """
    Manage indexing and queueing of messages.

    TODO: does `finalize` need to be made thread-safe?
    """
    get_deliverable_lock = threading.Lock()

    MESSAGE_INDEX = {}
    MESSAGE_DELIVERY_QUEUE = Queue.PriorityQueue()

    @classmethod
    def get(cls):
        with cls.get_deliverable_lock:
            msg = cls.MESSAGE_DELIVERY_QUEUE.get()
            if not msg.deliverable:
                cls.MESSAGE_DELIVERY_QUEUE.put(msg)
```

```
                raise NotDeliverable('Message {} at head of queue is not deliverable
yet.'
                                     .format(msg.msg_id))
        return msg

    @classmethod
    def put(cls, msg):
        cls.MESSAGE_INDEX[msg.msg_id] = msg
        return cls.MESSAGE_DELIVERY_QUEUE.put(msg)

    @classmethod
    def find(cls, msg_id):
        return cls.MESSAGE_INDEX.get(msg_id)

    @classmethod
    def finalize(cls, msg_id, timestamp):
        msg = cls.MESSAGE_INDEX.get(msg_id)
        if msg is None:
            raise ValueError('No message could be found with msg_id %s', msg_id)
        msg.timestamp = timestamp
        msg.status = MULTICAST_STATUS_FINAL
        msg.deliverable = True


class ClockedThreadingTCPServer(SocketServer.ThreadingTCPServer):
    """
    A TCP server that spawns threads when handling requests and increments a Lamport
clock
    for every received message.
    """
    allow_reuse_address = True

    def process_request(self, request, client_address):
        clock.tick()
        SocketServer.ThreadingTCPServer.process_request(self, request, client_address)
        return


class MulticastRequestHandler(SocketServer.StreamRequestHandler):
    def handle(self):
        msg = self.rfile.readline().strip()
        payload = json.loads(msg)
        parsed_msg = MulticastMessage(**payload)
        logger.debug('Received MulticastMessage %s', parsed_msg)

        clock.recv_msg(parsed_msg.timestamp)

        if parsed_msg.status == MULTICAST_STATUS_TENTATIVE:
            # Update timestamp to the latest timestamp after recv_msg
            parsed_msg.timestamp = clock.counter
            MulticastMessageQueue.put(parsed_msg)
            logger.debug('Responding with proposed timestamp %s', parsed_msg.timestamp)
            response = {
                'msg_id': parsed_msg.msg_id,
                'proposed_timestamp': parsed_msg.timestamp
            }
            self.write_response(response)
        elif parsed_msg.status == MULTICAST_STATUS_FINAL:
            logger.debug('Finalizing message %s', parsed_msg.msg_id)
            # Write finalized timestamp to previously enqueued msg
            try:
                MulticastMessageQueue.finalize(parsed_msg.msg_id, parsed_msg.timestamp)
            except ValueError:
                # For some reason the finalized message isn't in our queue, but since
it's
                # final we can go ahead and add it
                MulticastMessageQueue.put(parsed_msg)
            response = {
```

```
                'msg_id': parsed_msg.msg_id,
                'ack': True
            }
            self.write_response(response)

    def write_response(self, response):
        clock.tick()
        response['multicast_timestamp'] = clock.counter
        logger.debug('Writing serialized version of: %s', response)
        serialized_response = json.dumps(response)
        self.wfile.write(serialized_response + '\n')


class MulticastClient(object):
    def __init__(self, group_hosts, delimiter='\n'):
        """
        A client that knows how to propose and then finalize a message.

        :param group_hosts: a list of (HOST, PORT) tuples to multicast a message to
        """
        self.group_hosts = group_hosts
        self.delimiter = delimiter

    def remove_host(self, host):
        self.group_hosts.remove(host)

    def add_host(self, host):
        self.group_hosts.append(host)

    def send(self, msg):
        proposed_msg = MulticastMessage(msg)
        proposals = self._propose(proposed_msg)

        final_timestamp = max(proposals)
        proposed_msg.status = MULTICAST_STATUS_FINAL
        proposed_msg.timestamp = final_timestamp
        acks = self._finalize(proposed_msg)
        for ack in acks:
            if not ack:
                logger.warn('Got a non-OK response from a host!')

    def _send_to_group(self, msg):
        """
        Sends `msg` to all members of the group and returns their (raw) responses.
        """
        responses = []
        for host in self.group_hosts:
            logger.debug('Getting socket for host %s', host)
            try:
                client = socket.create_connection(host)
            except:
                logger.warn('Could not create connection with %s', host)
                continue
            client.sendall(msg.to_json() + self.delimiter)
            response = util.recv_until(client, self.delimiter)
            responses.append(json.loads(response))
        return responses

    def _propose(self, proposed_msg):
        raw_responses = self._send_to_group(proposed_msg)
        proposed_timestamps = []
        for response in raw_responses:
            assert response['msg_id'] == proposed_msg.msg_id
            proposed_timestamps.append(response['proposed_timestamp'])
        return proposed_timestamps

    def _finalize(self, msg):
        raw_responses = self._send_to_group(msg)
```

```
        acks = []
        for response in raw_responses:
            assert response['msg_id'] == msg.msg_id
            acks.append(response['ack'])
        return acks


class MulticastWorker(object):
    @classmethod
    def _watch_message(cls, handler):
        logger.debug("Waiting for a deliverable message...")
        counter = 0
        while True:
            try:
                msg = MulticastMessageQueue.get()
            except NotDeliverable as exc:
                logger.debug(exc.message)
            else:
                logger.debug('Got message, delivering it...')
                handler(msg)
            counter += 1
            logger.debug('Worker timer (NOT CLOCK) incremented to %s', counter)
        logger.debug('Worker exiting.')

    @classmethod
    def start_workers(cls, num_workers, message_handler):
        """
        Spawns threads that wait on deliverable messages and call `message_handler` with
        them.

        :param num_workers: number of worker threads to start
        :param message_handler: handler for deliverable messages
        """
        for i in xrange(num_workers):
            logger.debug('Starting worker thread %s', i)
            worker = threading.Thread(
                name='Worker {}'.format(i),
                target=cls._watch_message,
                args=(message_handler,)
            )
            worker.setDaemon(True)
            worker.start()


def multicast_serve(listen_addr, num_workers, msg_handler):
    """
    A convenience method to spin up worker threads and start a server listening for
    multicast messages.

    :param listen_host: the host to listen at
    :param listen_port: the port to listen at
    :param num_workers: the number of worker threads to spawn
    :param msg_handler: a handler for messages once they're found deliverable
    :return: None
    """
    # start listening for multicast messages
    server = util.ReusableThreadingTCPServer(listen_addr, MulticastRequestHandler)
    server_thread = threading.Thread(name='Server', target=server.serve_forever)
    server_thread.start()

    MulticastWorker.start_workers(num_workers, msg_handler)

    return server
```

protocol.py

```
import json
```

```
import logging

logger = logging.getLogger(__name__)


def node_message(fn):
    def inner(*args, **kwargs):
        val = fn(*args, **kwargs)
        val['client_type'] = 'node'
        return val
    return inner


class Protocol(object):
    # client types
    UNKNOWN_CLIENT = 'unknown client'
    EXTERNAL = 'external'
    NODE = 'node'

    READ_ALL = 'readall'
    READ = 'read'
    READ_RESPONSE = 'read OK'
    READ_ALL_RESPONSE = 'readall OK'
    UPDATE = 'update'
    ADD_INT = 'add int'

    # DataNodeApp entry sequence
    ENTER = 'enter'
    ENTRY_SYNC = 'sync'
    ENTRY_ACCEPTED = 'accepted'
    ENTRY_FINALIZE = 'finalize'
    ENTRY_FINALIZE_ACK = 'ack_finalize'

    @classmethod
    def serialize(cls, message):
        return json.dumps(message) + '\n'

    # Methods for operation on actual data structure.
    @classmethod
    def get_read(cls, index):
        msg = {
            'method': cls.READ,
            'index': index
        }
        return msg

    @classmethod
    def get_read_response(cls, group_id, index, data):
        msg = {
            'method': cls.READ_RESPONSE,
            'group_id': group_id,
            'index': index,
            'data': data,
        }
        return msg

    @classmethod
    def get_read_all_response(cls, group_id, data):
        msg = {
            'method': cls.READ_ALL_RESPONSE,
            'group_id': group_id,
            'data': data,
        }
        return msg


    # Methods for node entry/sync sequence.
    @classmethod
```

```
    def get_entry(cls, group_id, proc_id):
        msg = {
            'client_type': cls.NODE,
            'method': cls.ENTER,
            'group_id': group_id,
            'proc_id': proc_id,
        }
        return msg

    @classmethod
    def get_entry_finalization(cls, group_id, proc_id):
        msg = {
            'client_type': cls.NODE,
            'method': cls.ENTRY_FINALIZE,
            'group_id': group_id,
            'proc_id': proc_id,
        }
        return msg

    @classmethod
    def get_entry_accepted(cls, group_id, proc_id):
        msg = {
            'method': cls.ENTRY_ACCEPTED,
            'group_id': group_id,
            'proc_id': proc_id,
        }
        return msg

    @classmethod
    def get_ack_entry_finalization(cls, group_id, proc_id):
        msg = {
            'method': cls.ENTRY_FINALIZE_ACK,
            'group_id': group_id,
            'proc_id': proc_id,
        }
        return msg

    @classmethod
    def get_readall(cls, group_id):
        msg = {
            'method': cls.READ_ALL,
            'group_id': group_id,
        }
        return msg

    @classmethod
    def get_add_int(cls, group_id, value):
        msg = {
            'method': cls.ADD_INT,
            'group_id': group_id,
            'value': value,
        }
        return msg

    @classmethod
    def get_entry_sync(cls, group_id, proc_id, data):
        msg = {
            'method': cls.ENTRY_SYNC,
            'group_id': group_id,
            'proc_id': proc_id,
            'data': data,
        }
        return msg
```

util.py

```
"""
```

```
A collection of generic networking utilities.
"""
import logging
import SocketServer

logger = logging.getLogger(__name__)


def recv_until(sock, delimiter='\n'):
    """
    Receives a message until it sees `delimiter` and then returns the message (excluding
    the delimiter). Code derived from "Foundations of Python Network Programming", 2nd
    ed., listing 7-1.
    :param sock: a client socket
    :param delimiter: the delimiter to await, e.g. '\n'
    :return: the received message -- STILL SERIALIZED, excluding the delimiter
    """
    message = ''
    while not message.endswith(delimiter):
        data = sock.recv(4096)
        if not data:
            raise EOFError('socket closed before we saw a {!r}', delimiter)
        message += data
    message = message.rstrip(delimiter)
    logger.debug('Got message %s', message)
    return message


class ReusableThreadingTCPServer(SocketServer.ThreadingTCPServer):
    allow_reuse_address = True
```

Directory: profiling

fusion_comparison_fixed_f.py

```
import argparse
import datetime
import logging
import os
import shlex
import signal
import subprocess
import sys
import time

import collections

logging.basicConfig(level=logging.DEBUG)
logger = logging.getLogger(__name__)

NODE_CONFIG_FILE = 'node_config_f.ini'
CLIENT_CMD_TMPL = 'python -m fault_tolerance_eval.data_node.app examples/{} {}'
CORRECTNESS_SCRIPT_NAME = 'test_correctness_fixed_f.py'


def launch_nodes(num_nodes):
    logger.debug('Launching %s nodes...', num_nodes)

    # bit of a hack for making sure we're crashing/recovering nodes group-by-group
    GROUPS = {
        1: 'A',
        2: 'A',
        3: 'B',
        4: 'B',
        5: 'C',
        6: 'C',
```

```
        7: 'D',
        8: 'D',
        9: 'E',
        10: 'E',
    }

    # this is a mapping from {<group id>: {<process>: <config section ID>, ... }}
    node_processes = collections.defaultdict(dict)

    # xrange(start, end) is exclusive of end
    for i in xrange(1, num_nodes + 1):
        logger.info('Starting node %s', i)
        cmd_args = shlex.split(CLIENT_CMD_TMPL.format(NODE_CONFIG_FILE, i))

        node_process = subprocess.Popen(cmd_args)

        # Record node section ID for this process
        node_processes[GROUPS[i]][node_process] = i
        time.sleep(2)

    logger.debug('Launched nodes: %s', node_processes)
    return node_processes


def kill_node_processes(processes, recover=True):
    new_processes = collections.defaultdict(dict)

    for group, node_mapping in processes.iteritems():
        for p, node_section_id in node_mapping.iteritems():
            logger.warn('Murdering process %s, node section %s', p.pid, node_section_id)
            os.kill(p.pid, signal.SIGINT)

            if recover:
                time.sleep(3)
                logger.warn('Restarting node %s', node_section_id)
                client_cmd = shlex.split(CLIENT_CMD_TMPL.format(NODE_CONFIG_FILE,
                                                                node_section_id))
                new_process = subprocess.Popen(client_cmd)
                time.sleep(3)
                new_processes[group][new_process] = node_section_id

    logger.debug('Launched new processes: %s', new_processes)

    return new_processes


if __name__ == '__main__':
    comparison_start = datetime.datetime.now()
    logger.info('Beginning comparison at %s', comparison_start)
    comparison_complete = False

    # Parse command line arguments
    parser = argparse.ArgumentParser(
        description='Do a run of this active replication example system for comparison '
                    'with a similar fusion-based system\'s run.')
    parser.add_argument('-f', '--fault_tolerance', type=int, default=1,
                        help='Number of crash faults to tolerate.')
    parser.add_argument('-g', '--groups', type=int, default=1,
                        help='Number of replication groups to support.')
    parser.add_argument('-o', '--ops', type=int, default=100,
                        help='Number of operations per primary. All operations are '
                             'appends.')
    parser.add_argument('-s', '--step', type=int, default=10,
                        help='Interval at which to stop and recover all primaries.')
    args = parser.parse_args()

    # For each group, we're going to launch f+1 nodes
    num_nodes = args.groups * (args.fault_tolerance + 1)
```

```
    logger.info('Starting group manager...')
    cmd = 'python -m fault_tolerance_eval.group_manager.app localhost 9999 {} ' \
          'examples/{}'.format(num_nodes, NODE_CONFIG_FILE)
    cmd_args = shlex.split(cmd)
    group_manager_process = subprocess.Popen(cmd_args)
    logger.debug('Server process ID: %s', group_manager_process.pid)

    # Give group manager time to set up
    time.sleep(5)

    node_processes = launch_nodes(num_nodes)

    # Give all nodes time to set up
    time.sleep(5)

    ops_performed = 0
    ops_cmd_tmpl = 'python examples/{} {} {}'

    while ops_performed < args.ops:
        # Synchronously call test script and wait on its completion
        logger.info('Calling %s', CORRECTNESS_SCRIPT_NAME)
        subprocess.call(shlex.split(ops_cmd_tmpl.format(CORRECTNESS_SCRIPT_NAME,
args.step, args.groups)))
        ops_performed += args.step
        node_processes = kill_node_processes(node_processes)

    if ops_performed >= args.ops:
        comparison_complete = True

    # Stay alive so ctrl-C gets passed to child processes
    try:
        while True:
            if comparison_complete:
                comparison_end = datetime.datetime.now()
                comparison_duration = comparison_end - comparison_start
                logger.info('Ending comparison at %s. Seconds elapsed: %s',
comparison_end,
                    comparison_duration.total_seconds())
                kill_node_processes(node_processes, False)
                os.kill(group_manager_process.pid, signal.SIGINT)
                sys.exit(0)
            time.sleep(5)
    except KeyboardInterrupt:
        logger.debug('Got interrupt, wrapping up.')
```

loggers.py

```
"""
Set up several loggers for use during profiling.
"""
import logging

csv_formatter =
logging.Formatter('%(process)d,"%(asctime)s",%(name)s,%(levelname)s,%(message)s')


def get_backup_size_logger_for_node(group_id, node_id):
    # If multiple processes are running for the same node ID, this is an issue. But then
    # you have other issues anyway.
    filename = '{group_id}-{node_id}_backup_size_log.csv'.format(group_id=group_id,
                                                                 node_id=node_id)
    backup_size_handler = logging.FileHandler(filename)
    backup_size_handler.setFormatter(csv_formatter)

    backup_size_logger = logging.getLogger('backup_size')
    backup_size_logger.addHandler(backup_size_handler)
```

```
    return backup_size_logger

MULTICAST_MSG_COUNT_FILENAME = 'multicast_msg_count_log.csv'
multicast_msg_count_handler = logging.FileHandler(MULTICAST_MSG_COUNT_FILENAME)
multicast_msg_count_handler.setFormatter(csv_formatter)

multicast_msg_count_logger = logging.getLogger('multicast_msg_count')
multicast_msg_count_logger.addHandler(multicast_msg_count_handler)

UPDATE_TIME_FILENAME = 'update_time_log.csv'
update_time_handler = logging.FileHandler(UPDATE_TIME_FILENAME)
update_time_handler.setFormatter(csv_formatter)

update_time_logger = logging.getLogger('update_time')
update_time_logger.addHandler(update_time_handler)

RECOVERY_TIME_FILENAME = 'recovery_time.csv'
recovery_time_handler = logging.FileHandler(RECOVERY_TIME_FILENAME)
recovery_time_handler.setFormatter(csv_formatter)

recovery_time_logger = logging.getLogger('recovery_time')
recovery_time_logger.addHandler(recovery_time_handler)
```

size.py

```
from __future__ import print_function
from sys import getsizeof, stderr
from itertools import chain
from collections import deque


def total_size(o, handlers=None, verbose=False):
    """ Returns the approximate memory footprint an object and all of its contents.

    Automatically finds the contents of the following builtin containers and
    their subclasses:  tuple, list, deque, dict, set and frozenset.
    To search other containers, add handlers to iterate over their contents:

        handlers = {SomeContainerClass: iter,
                    OtherContainerClass: OtherContainerClass.get_elements}

    Modified from recipe found at: http://code.activestate.com/recipes/577504/
    """
    dict_handler = lambda d: chain.from_iterable(d.items())
    all_handlers = {tuple: iter,
                    list: iter,
                    deque: iter,
                    dict: dict_handler,
                    set: iter,
                    frozenset: iter,
                   }

    if handlers is None:
        handlers = {}

    all_handlers.update(handlers)     # user handlers take precedence
    seen = set()                      # track which object id's have already been seen
    default_size = getsizeof(0)       # estimate sizeof object without __sizeof__

    def sizeof(obj):
        if id(obj) in seen:       # do not double count the same object
            return 0
        seen.add(id(obj))
        s = getsizeof(obj, default_size)

        if verbose:
            print(s, type(obj), repr(obj), file=stderr)
```

```
        for typ, handler in all_handlers.items():
            if isinstance(obj, typ):
                s += sum(map(sizeof, handler(obj)))
                break
        return s

    return sizeof(o)

##### Example calls #####

if __name__ == '__main__':
    d = dict(a=1, b=2, c=3, d=[4,5,6,7], e='a string of chars')
    print(total_size(d, verbose=True))

    l = []
    print(total_size(l, verbose=True))

    l = [1]
    print(total_size(l, verbose=True))

    l = [1, 2, 3, 4]
    print(total_size(l, verbose=True))

    l = [1] * 1000
    print(total_size(l))
```