The State Machine Approach '08

Bryan Turner
Feb, 2008

Submitted to Wikipedia.org as an extension to the article “State machine replication”.

Introduction

> *Introduction from Schneider’s 1990 survey:*
> “Distributed software is often structured in terms of *clients* and *services*. Each service comprises one or more *servers* and exports *operations* that clients invoke by making *requests*. Although using a single, centralized, server is the simplest way to implement a service, the resulting service can only be as fault tolerant as the processor executing that server. If this level of fault tolerance is unacceptable, then multiple servers that fail independently must be used. Usually, replicas of a single server are executed on separate processors of a distributed system, and protocols are used to coordinate client interactions with these replicas. The physical and electrical isolation of processors in a distributed system ensures that server failures are independent, as required.
>
> “The state machine approach is a general method for implementing a fault-tolerant service by replicating servers and coordinating client interactions with server replicas. The approach also provides a framework for understanding and designing replication management protocols. Many protocols that involve replication of data or software - be it for masking failures or simply to facilitate cooperation without centralized control - can be derived using the state machine approach. Although few of these protocols actually were obtained in this manner, viewing them in terms of state machines helps in understanding how and why they work.” [1]

Preliminaries

State Machine Definition

For the subsequent discussion a **State Machine** will be defined as the following tuple of values [2]:

- A set of **States**
- A set of **Inputs**
- A set of **Outputs**
- A transition function (Input x State -> State)
- An output function (Input x State -> Output)

- A distinguished State called Start.

A State Machine begins at the State labeled Start. Each Input received is passed through the transition and output function to produce a new State and an Output. The State is held stable until a new Input is received, while the Output is communicated to the appropriate receiver.

It should be clear that any algorithm can be implemented using this model if driven by an appropriate Input stream. In particular, this discussion requires a State Machine to have the following property:

> **Deterministic:**
> Multiple copies of the same State Machine begun in the Start state, and receiving the same Inputs in the same order will arrive at the same State having generated the same Outputs.

Fault Tolerance Explained

Determinism is an ideal characteristic for providing fault-tolerance. Intuitively, if multiple copies of a system exist, a fault in one would be noticeable as a difference in the State or Output from the others.

A little deduction shows the minimum number of copies needed for fault-tolerance is three; one which has a fault, and two others to whom we compare State and Output. Two copies is not enough; there is no way to tell which copy is the faulty one.

Further deduction shows a three-copy system can support at most one failure (after which it must repair or replace the faulty copy). If more than one of the copies were to fail, all three States and Outputs might differ, and there would be no way to choose which is the correct one.

Research has shown [3] that in general a system which supports F failures must have 2F+1 copies (also called replicas). The extra copies are used as evidence to decide which of the copies are correct and which are faulty. Special cases can improve these bounds [4].

All of this deduction pre-supposes that replicas are experiencing only random independent faults such as memory errors or hard-drive crash. Failures caused by replicas which attempt to lie, deceive, or collude are called Byzantine Failures [5]. Both random and byzantine failures are supported by the State Machine Approach, with isolated changes.

It should be noted that failed replicas are not required to stop; they may continue operating, including generating spurious or incorrect Outputs.

The State Machine Approach

The preceding intuitive discussion implies a simple technique for implementing a fault-tolerant service in terms of a State Machine [1][2][16]:

1) Place copies of the State Machine on multiple, independent servers.
2) Receive client requests, interpreted as Inputs to the State Machine.
3) Choose an ordering for the Inputs.
4) Execute Inputs in the chosen order on each server.
5) Respond to clients with the Output from the State Machine.
6) Monitor replicas for differences in State or Output.

The remainder of this article develops the details of this technique.

Step 1 and 2 are outside the scope of this article.
Step 3 is the critical operation, see Ordering Inputs.
Step 4 is covered by the State Machine Definition.
Step 5, see Sending Outputs.
Step 6, see Auditing and Failure Detection.

The appendix contains discussion on typical extensions used in real-world systems such as Logging, Checkpoints, Reconfiguration, and State Transfer.

Ordering Inputs

The critical step in building a distributed system of State Machines is choosing an order for the Inputs to be processed. Since all non-faulty replicas will arrive at the same State and Output if given the same Inputs, it is imperative that the Inputs are submitted in an equivalent order at each replica. Many solutions have been proposed in the literature [2][6][7][8].

A **Visible Channel** is a communication path between two entities actively participating in the system (such as clients and servers).
Example: client to server, server to server

A **Hidden Channel** is a communication path which is not revealed to the system.
Example: client to client channels are usually hidden; such as users communicating over a telephone, or a process writing files to disk which are read by another process.

When all communication paths are visible channels and no hidden channels exist, a partial global order (**Causal Order**) may be inferred from the pattern of communications [9][8]. Causal Order may be derived independently by each server. Inputs to the State Machine may be executed in Causal Order, guaranteeing consistent State and Output for all non-faulty replicas.

In open systems, hidden channels are common and a weaker form of ordering must be used. An order of Inputs may be defined using a voting protocol whose results depend only on the visible channels.

The problem of voting for a *single* value by a group of independent entities is called **Consensus**. By extension, a *series* of values may be chosen by a series of consensus instances. This problem becomes difficult when the participants or their communication medium may experience failures [10].

Inputs may be ordered by their position in the series of consensus instances (**Consensus Order**)[7]. Consensus Order may be derived independently by each server. Inputs to the State Machine may be executed in Consensus Order, guaranteeing consistent State and Output for all non-faulty replicas.

Optimizing Causal & Consensus Ordering

In some cases additional information is available (such as real-time clocks). In these cases, it is possible to achieve more efficient causal or consensus ordering for the Inputs, with a reduced number of messages, fewer message rounds, or smaller message sizes. See references for details [6][1][11][4]

Further optimizations are available when the semantics of State Machine operations are accounted for (such as Read vs Write operations). See references [2][12].

Sending Outputs

Client requests are interpreted as Inputs to the State Machine, and processed into Outputs in the appropriate order. Each replica will generate an Output independently. Non-faulty replicas will always produce the same Output. Before the client response can be sent, faulty Outputs must be filtered out. Typically, a majority of the Replicas will return the same Output, and this Output is sent as the response to the client.

System Failure

If there is no majority of replicas with the same Output, or if less than a majority of replicas returns an Output, a system failure has occurred. The client response must be the unique Output: **FAIL**.

Auditing and Failure Detection

The permanent, unplanned compromise of a replica is called a **Failure**. Proof of failure is difficult to obtain, as the replica may simply be slow to respond [13], or even lie about its status [5].

Non-faulty replicas will always contain the same State and produce the same Outputs. This invariant enables failure detection by comparing States and Outputs of all replicas. Typically, a replica with State or Output which differs from the majority of replicas is declared faulty.

A common implementation is to pass checksums of the current replica State and recent Outputs among servers. An Audit process at each server restarts the local replica if a deviation is detected [14]. Cryptographic security is not required for checksums.

It is possible that the local server is compromised, or that the Audit process is faulty, and the replica continues to operate incorrectly. This case is handled safely by the Output filter described previously (see Sending Outputs).

Appendix: Extensions

Input Log

In a system with no failures, the Inputs may be discarded after being processed by the State Machine. Realistic deployments must compensate for transient non-failure behaviors of the system such as message loss, network partitions, and slow processors [14].

One technique is to store the series of Inputs in a log. During times of transient behavior, replicas may request copies of a log entry from another replica in order to fill in missing Inputs [7].

In general the log is not required to be persistent (it may be held in memory). A persistent log may compensate for extended transient periods, or support additional system features such as Checkpoints, and Reconfiguration.

Checkpoints

If left unchecked a log will grow until it exhausts all available storage resources. For continued operation, it is necessary to forget log entries. In general a log entry may be forgotten when its contents are no longer relevant (for instance if all replicas have processed an Input, the knowledge of the Input is no longer needed).

A common technique to control log size is store a duplicate State (called a **Checkpoint)**, then discard any log entries which contributed to the checkpoint. This saves space when the duplicated State is smaller than the size of the log.

Checkpoints may be added to any State Machine by supporting an additional Input called **CHECKPOINT**. Each replica maintains a checkpoint in addition to the current State value. When the log grows large, a replica submits the CHECKPOINT command just like a client request. The system will ensure non-faulty replicas process this command in the same order, after which all log entries before the checkpoint may be discarded.

In a system with checkpoints, requests for log entries occurring before the checkpoint are ignored. Replicas which cannot locate copies of a needed log entry are faulty and must re-join the system (see Reconfiguration).

Reconfiguration

Reconfiguration allows replicas to be added and removed from a system while client requests continue to be processed. Planned maintenance and replica failure are common examples of reconfiguration.

Quitting

When a server detects its State or Output is faulty (see Auditing and Failure Detection), it may selectively exit the system. Likewise, an administrator may manually execute a command to remove a replica for maintenance.

A new Input is added to the State Machine called **QUIT**[2][6]. A replica submits this command to the system just like a client request. All non-faulty replicas remove the quitting replica from the system upon processing this Input. During this time, the replica may ignore all protocol messages. If a majority of non-faulty replicas remain, the quit is successful. If not, there is a System Failure.

Joining

After quitting, a failed server may selectively **restart** or **re-join** the system. Likewise, an administrator may add a new replica to the group for additional capacity.

A new Input is added to the State Machine called **JOIN**. A replica submits this command to the system just like a client request. All non-faulty replicas add the joining node to the system upon processing this Input. A new replica must be up-to-date on the system's State before joining (see State Transfer).

State Transfer

When a new replica is made available or an old replica is restarted, it must be brought up to the current State before processing Inputs (see Joining). Logically, this requires applying every Input from the dawn of the system in the appropriate order.

Typical deployments short-circuit the logical flow by performing a **State Transfer** of the most recent Checkpoint (see Checkpoints). This involves directly copying the State of one replica to another using an out-of-band protocol.

A checkpoint may be large, requiring an extended transfer period. During this time, new Inputs may be added to the log. If this occurs, the new replica must also receive the new Inputs and apply them after the checkpoint is received. Typical deployments add the new replica as an observer to the ordering protocol before beginning the state transfer, allowing the new replica to collect Inputs during this period.

Optimizing State Transfer

Common deployments reduce state transfer times by sending only State components which differ. This requires knowledge of the State Machine internals. Since state transfer is usually an out-of-band protocol, this assumption is not difficult to achieve.

Compression is another feature commonly added to state transfer protocols, reducing the size of the total transfer.

Leader Election (for Paxos)

Paxos[7] is a protocol for solving consensus, and may be used as the protocol for implementing Consensus Order.

Paxos requires a single leader to ensure liveness [7]. That is, one of the replicas must remain leader long enough to achieve consensus on the next operation of the state machine. System behavior is unaffected if the leader changes after every instance, or if the leader changes multiple times per instance. The only requirement is that one replica remain leader long enough to move the system forward.

Conflict Resolution

In general, a leader is necessary only when there is disagreement about which operation to perform [11], and if those operations conflict in some way (for instance, if they do not commute) [12].

When conflicting operations are proposed, the leader acts as the single authority to set the record straight, defining an order for the operations, allowing the system to make progress.

With Paxos, multiple replicas may believe they are leaders at the same time. This property makes Leader Election for Paxos very simple, and any algorithm which guarantees an 'eventual leader' will work.

References

[1] Schneider, Fred (1990) "Implementing Fault-Tolerant Services Using the State Machine Approach: A Tutorial" ACM Computing Surveys 22
http://www.eecs.harvard.edu/cs262/DSbook.c7.pdf

[2] Lamport, Leslie (1978) "The Implementation of Reliable Distributed Multiprocess Systems" Computer Networks Vol. 2 pp. 95-114
http://research.microsoft.com/users/lamport/pubs/pubs.html#implementation

[3] Lamport, Leslie (2004) "Lower Bounds for Asynchronous Consensus
http://research.microsoft.com/users/lamport/pubs/pubs.html#lower-bound

[4] Lamport, Leslie; Massa, Mike (2004) "Cheap Paxos" Proceedings of the International Conference on Dependable Systems and Networks (DSN 2004)
http://research.microsoft.com/users/lamport/pubs/pubs.html#web-dsn-submission

[5] Lamport, Leslie; Shostak, Robert; Pease, Marshall (1982) "The Byzantine Generals Problem" ACM Transactions on Programming Languages and Systems Vol. 4,3 pp. 382–401
http://research.microsoft.com/users/lamport/pubs/pubs.html#byz

[6] Lamport, Leslie (1984) "Using Time Instead of Timeout for Fault-Tolerant Distributed Systems" ACM Transactions on Programming Languages and Systems Vol. 6,2 pp. 254-280
http://research.microsoft.com/users/lamport/pubs/pubs.html#using-time

[7] Lamport, Leslie (1998) "The Part-Time Parliament" ACM Transactions on Computer Systems Vol. 16,2 pp. 133–169
http://research.microsoft.com/users/lamport/pubs/pubs.html#lamport-paxos

[8] Birman, Kenneth; Joseph, Thomas (1987) "Exploiting virtual synchrony in distributed systems" Proceedings of the 11th ACM Symposium on Operating systems principles (SOSP)
http://portal.acm.org/citation.cfm?id=37515&dl=ACM&coll=GUIDE

[9] Lamport, Leslie (1978) "Time, Clocks and the Ordering of Events in a Distributed System" Communications of the ACM Vol. 21,7 pp. 558–565
http://research.microsoft.com/users/lamport/pubs/pubs.html#time-clocks

[10] Lamport, Leslie (2004) "Lower Bounds for Asynchronous Consensus"
http://research.microsoft.com/users/lamport/pubs/pubs.html#lower-bound

[11] Lamport, Leslie (2005) “Fast Paxos”
http://research.microsoft.com/users/lamport/pubs/pubs.html#fast-paxos

[12] Lamport, Leslie (2005) “Generalized Consensus and Paxos”
http://research.microsoft.com/users/lamport/pubs/pubs.html#generalized

[13] Fischer, Michael (1985) “Impossibility of Distributed Consensus with One Faulty Process” Journal of the Association for Computing Machinery Vol. 32,2 pp. 347-382
http://research.microsoft.com/users/lamport/pubs/pubs.html#using-time

[14] Chandra, Tushar; Griesemer, Robert; Redstone, Joshua (2007) “Paxos Made Live – An Engineering Perspective” PODC '07: 26th ACM Symposium on Principles of Distributed Computing
http://labs.google.com/papers/paxos_made_live.html

[15] Castro, Miguel (2001) “Practical Byzantine Fault Tolerance”
http://citeseer.ist.psu.edu/castro01practical.html

[16] Lampson, Butler (1996) “How to Build a Highly Available System Using Consensus”
http://research.microsoft.com/lampson/58-Consensus/Abstract.html