Asynchronous Consensus: A Model in TLA+

Stephen Tarzia

January 5, 2007

Abstract

The main goal of this project was to learn the TLA+ specification language and to start exploring the practical limitations of modeling and simulating unreliable systems. A TLA+ specification for a naive asynchronous consensus protocol is presented along with the model-checking times for one to eight processes under crash-free conditions. Model-Checking times increase drastically with added processes, making large-scale model-checking intractable without further model abstraction. The protocol is broken by adding a crash action, and preliminary analysis of failure detector modeling is given.

1 Introduction

1.1 Consensus

Consensus is a basic problem in distributed systems. Generally speaking, consensus is attained when a group (system) of independent entities (processes) negotiates to reach a unanimous decision. A solution to this problem is an algorithm each process can follow to ensure that consensus is always reached. We say that a distributed system is asynchronous if message transmission times and relative processor speeds are both finite but unbounded. We say that a system is unreliable if process crashes can occur randomly. A crashed process never changes state or responds to external events. It has been shown that the consensus problem is unsolvable in unreliable asynchronous systems [3]. Intuitively, this stems from the inability of a process to decide whether an unresponsive neighbor has crashed or is just very slow.

1.2 TLA+

TLA+ is a system specification language [5]. Formally, a system is a set of behaviors, a behavior is a sequence of states, and a state is an assignment of values to a set of variables. The primary advantage of specification is that it allows one to use a model-checker to simulate all possible behaviors of the system and thus identify any behaviors that reach unexpected states. TLC is the TLA+ model-checker.

A TLA+ code defines a set of variables, a set of actions that modify the variables, and a set of predicates on the variables. A TLC configuration file completes a system

specification by indicating the semantics of the TLA+ code. The configuration file identifies the initial state predicate, the next state action, and the temporal predicate. It also has model-checker directives that specify specific values to be assigned to constants, and predicates to be tested by the model checker.

1.3 Goals

My main task was to model the asynchronous consensus problem in TLA+ and verify it with the TLC model-checker. This task fits into a larger project on the design of reliable systems using unrealiable components. One of the goals of this project is to establish a design methodology for unreliable systems that incorporates specification and model-checking. My experience modeling consensus algorithms will be used as a (preliminary) case study in the Science of Design investigation. Consensus represents perhaps the most basic meaningful algorithm applicable to unreliable distributed systems.

2 Method

2.1 Specification Design Decisions

Abstraction There are many choices to make when specifying a system. A single system can be specified innumerable ways using different state representations and action granularities. Choice of state representation (variables) is somewhat trivial, having only cosmetic effects (here, the goal should be semantic clarity). Smaller action granularity increases the search depth (the maximal behavior length) and the branching factor (the number of possible next states) leading to much larger behavior search space and consequently longer model-checking time. It will also take more effort for the designer to write a finer-grained specification. Generally, there is a balance to strike between accuracy and behavior space size. This is a very difficult aspect of specification; a composite (multi-step) action can be specified atomically only if we are certain that interleaving other actions with it will not lead to an unexpected state. Thus, in the asynchronous consensus specification atomicity of certain actions was used to reduce behavior space size.

Symmetry Another method of behavior space reduction is the use of symmetry. This is done using the TLC module's Permutations function and the configuration file `SYMMETRY` directive. Again, care must be taken when using symmetry to ensure that permutation of a set in a behavior does not change the results. When symmetry on a set is invoked, states that differ only by permutation of that set are considered redundant; thus, the behavior space size is reduced. In the asynchronous consensus specification, symmetry was invoked on the processes and on the decision domain. We can be sure that this symmetry is permissible because the specification never references a particular process or decision value (it uses only existential and universal quantification). While symmetry does dramatically reduce the number of states considered, there is some overhead associated with the symmetry checking.

Therefore, it is possible that careless use of symmetry can actually increase model-checking run-time.
Symmetry reduced the number of initial states to precisely the number of processes in the system; there is one state for each possible number of processes that have the same candidate value (*pInitialValue[p]*) as the coordinator.

Problem simplification The third method of behavior space reduction is problem simplification. A binary decision domain of attack, retreat was chosen rather than allowing each process its own unique proposition value [6]. The consensus problem on any finite decision domain can be reduced to $\log(n)$ instances of the binary consensus problem where n is the size of the decision domain and each binary consensus instance represents deciding one bit of the final value. Since our interest in this problem is only theoretical (we are not modeling any specific consensus system) we accept the binary problem as an equally important form for which analysis is tractable.

Crash model Perhaps the most important design decision that I made was in the crash model. In the specification, the function *pIsCrashed[p]* denotes whether or not a process p has crashed. A process may only act if it has not crashed. Crashing is itself an action that an uncrashed process may take. The crash actions are not included in the weak fairness condition. Therefore, even though a process p always *can* crash it does not eventually *have* to. The result is that we can consider behaviors for which some processes never crash.

Liveness conditions Precisely speaking, consensus is the assignment of decision values to processes satisfying both *agreement* and *validity* conditions. Agreement requires that all processes decide on the same value. Validity requires that some process "proposed" the decided value (each process has a candidate value stored in *pInitialValue[p]*). When checking each of these conditions crashed processes are ignored if, and only if, they did not reach a decision prior to crashing.

Deadlock The TLC model-checker normally reports an error upon reaching a state wherein no actions are enabled (no valid next-state exists). The specification does bring the system to such a deadlock state when the consensus algorithm is complete, so I have disabled deadlock errors in my model-checking using the `-deadlock` flag.

2.2 The Naive Consensus Algorithm

- One process is assumed to be the coordinator. Assuming that each process has a numerical identifier, we can implement this by making process 0 act as the coordinator. (The specification chooses *any* process.)
- Coordinator broadcasts its initial value as a decision proposal and decides on this value itself.

- All other processes wait for the coordinator's proposal. When a process receives a proposal, it decides on the proposed value and sends an acknowledgment message back to the coordinator. This acknowledgment message is unnecessary for protocol correctness, but it is included in order to make the model-checking behavior search space larger (more closely resembling a real, useful, protocol).

2.3 Environment

The test machine had an Intel Core 2 6300 CPU @ 1.86 GHz with 1 GB RAM running a Linux 2.6.18 kernel. The Sun Java 2 VM version 1.5.0_09 was used. No significant memory swapping occurred during model-checking. TLC was invoked with the following syntax:

```
java -Xmx3500m tlc.TLC -deadlock -workers 2 -config [config_file] [tla_file]
```

3 Results

Number of Processes	States generated	Distinct states	Search depth	Model-Checking time[1]	
1	11	8	5	0m	0.596s
2	112	64	8	0m	0.696s
3	706	332	11	0m	0.900s
4	3344	1362	14	0m	2.092s
5	13172	4817	17	0m	22.609s
6	45722	15392	20	8m	13.847s
7	144909	45709	23	203m	5.954s
8	429112	128476	26	4424m	41.992s

The worst case probability that TLC did not check all states was for the 8 state model-checking where the probability was 2.0938389225580445E-9. New states are checked for distinctness by comparing a 64-bit hash value called a fingerprint with the fingerprints of all previously seen states. A collision in the hash function can cause a new distinct state to be treated as a recurrence of an old state, resulting in the search tree being erroneously pruned.

4 Discussion

The results of the consensus algorithm model-checking give several insights.

- Model-checking a medium or large sized system explicitly is intractable, even for very simple systems such as this consensus protocol. However, intuitively, positive results for up to eight processes indicate that the protocol is correct for any number of processes, since there does not seem to be any real change in

[1] user time as measured by the OS via the `time` command

the problem beyond three processes. Formally, this could be proved manually in the style of an induction proof (the model-checker is handling the base case and we do the induction case manually). In this case however, the protocol is so simple that a complete manual proof is trivial[2]; induction is not needed.

- It appears that the addition of a crash model does not create any significant hurdles for model checking. The branching factor is increased, but only moderately. This effect would also be seen if the protocol was made more complex by adding another action.

5 Future Work

This project can move in several directions:

- We can explore modeling of failure detectors [2]. No practical failure detector can provide perfect knowledge of the state of all processes. Thus, given a set of crashed and uncrashed processes, there are many possible valid states for its failure detectors. Modeling the failure detectors becomes a very challenging problem because it potentially involves a branching factor equal to the cardinality of the power set of processes (2^p). Once failure detectors are modeled, a correct asynchronous consensus algorithm can be specified. See appendix for more on failure detectors.

- We can try to extend the size of the systems we can check by applying either data abstraction or network invariants.

A Asynchronous consensus with failure TLA+ code

MODULE *AsyncConsensus_withFailure*

This module defines a consensus protocol for an asynchronous system
It follows the model proposed by *Chandra* and *Toueg*, 1991
- Any pair of processes can communicate (complete graph topology)
- Communication channel is reliable and FIFO
- *Processes* have a priori knowledge of every process's unique identifier
- Relative processor speeds and message transmission times are finite but unbounded
- Process stop-failures occur as actions. As many as n such crashes can occur in a run.

EXTENDS *Naturals*, *Sequences*, *TLC*, *FiniteSets*
CONSTANTS *Processes*, the set of all processes in the network
Domain the set of all possible decision values
$Perms \triangleq Permutations(Processes) \cup Permutations(Domain)$ we use symmetry on the processes and the domain to simplify the initial state space

[2]Agreement and Validity are satisfied because all processes decide on the coordinator's initial value. There are no loops, so Termination is trivial.

VARIABLES

VARIABLES $pMsgQ$, $pInitialValue$, $pDecisionValue$, $pIsCrashed$, $Coordinator$, $pProtocolState$

$pMsgQ[n]$ is state of process n's communication system.

Its value is a *FIFO* queue of messages from other nodes to be delivered to this node

$pInitial[n]$ is the inital value of process n.

$pDecisionValue[n]$ is the decision value of process n. This is set when process n has reached a decision

$pIsCrashed[n]$ indicates whether or not process n is crashed.

Coordinator is the process that makes a proposal

$pProtocolState[n]$ is the protocol state of process n.

$State \triangleq \langle pMsgQ, pInitialValue, pDecisionValue, pIsCrashed, Coordinator, pProtocolState \rangle$

$Messages \triangleq$

Defines the set of all possible messages

$[type : \{\text{“proposal”}, \text{“ack”}, \text{“nack”}\},$
$src : Processes,$
$dst : Processes,$
$data : Domain \cup \{\text{“”}\}]$

PREDICATES

$Init \triangleq$

Initial state

$\land\ pDecisionValue = [p \in Processes \mapsto \text{“”}]$
$\land\ pInitialValue \in [Processes \to Domain]$
$\land\ pMsgQ = [p \in Processes \mapsto \langle\rangle]$
$\land\ pIsCrashed = [p \in Processes \mapsto \text{FALSE}]$
$\land\ Coordinator \in Processes$
$\land\ pProtocolState = [p \in Processes \mapsto \text{“awaitingProposal”}]$

$TypeOK \triangleq$

Type invariant predicate

$\land\ pMsgQ \in [Processes \to Seq(Messages)]$
$\land\ pDecisionValue \in [Processes \to Domain \cup \{\text{“”}\}]$ “” indicates not decided
$\land\ pInitialValue \in [Processes \to Domain]$
$\land\ pIsCrashed \in [Processes \to \text{BOOLEAN}\ ]$
$\land\ Coordinator \in Processes$
$\land\ pProtocolState \in [Processes \to \{\text{“awaitingProposal”}, \text{“seenProposal”}, \text{“decided”}, \text{“awaitingAck”}\}]$

ACTIONS

$Send(msg) \triangleq$

Message is sent. Source and destination processes are $msg.src$ and $msg.dst$

$\land\ msg \in Messages$ enabled if msg is valid
$\land\ pMsgQ' = [pMsgQ \text{ EXCEPT } ![msg.dst] = Append(pMsgQ[msg.dst],\ msg)]$
append the message to the $msgQ$ of the destination process

$NewMsg(m) \triangleq$
Process $m.dst$ handles the receipt of a new message m
CASE $m.type =$ "proposal" $\rightarrow$
The message received was a proposal
$\land\ pDecisionValue' = [pDecisionValue \text{ EXCEPT } ![m.dst] = m.data]$
receiving process decides on the value received
$\land\ pProtocolState' = [pProtocolState \text{ EXCEPT } ![m.dst] =$ "seenProposal"$]$
change state to reflect seeing the proposal
$\square$OTHER $\rightarrow$ UNCHANGED $\langle pDecisionValue,\ pProtocolState\rangle$

$Deliver(p) \triangleq$
Message is moved from a process p's $msgQ$ into its deliver buffer and delivered flag is set to signal the process that it has received a new message.
$\land\ pMsgQ[p] \neq \langle\rangle$ enabled if $msgQ$ is not empty
$\land\ NewMsg(Head(pMsgQ[p]))$ the new message handling action
$\land\ pMsgQ' = [pMsgQ \text{ EXCEPT } ![p] = Tail(pMsgQ[p]\ )]$
pop message off $msgQ$
$\land$ UNCHANGED $pIsCrashed$

$Broadcast(bType,\ bSrc,\ bData) \triangleq$
Reliable boadcast of a message from src to all processes
$pMsgQ' = [p \in Processes \mapsto Append(pMsgQ[p],$
$[type \mapsto bType,\ src \mapsto bSrc,\ dst \mapsto p,\ data \mapsto bData])]$
append the new message to all of the message queues

$Crash(p) \triangleq$
Process crash action
$\land\ pIsCrashed' = [pIsCrashed \text{ EXCEPT } ![p] = \text{TRUE}]$
$\land$ UNCHANGED $\langle pMsgQ,\ pProtocolState,\ pDecisionValue\rangle$

$Proposal(p) \triangleq$
Process p proposes its own initial value
$\land\ Broadcast($"proposal"$,\ p,\ pInitialValue[p])$
$\land\ pProtocolState' = [pProtocolState \text{ EXCEPT } ![p] =$ "seenProposal"$]$ p knows what it proposed
$\land\ pDecisionValue' = [pDecisionValue \text{ EXCEPT } ![p] = pInitialValue[p]]$ decide on $InitialValue$

$Ack(p) \triangleq$
Process p acknowledges receipt of a proposal
$\land\ Send([type \mapsto$ "ack"$,\ src \mapsto p,\ dst \mapsto Coordinator,\ data \mapsto$ ""$])$
$\land\ pProtocolState' = [pProtocolState \text{ EXCEPT } ![p] =$ "decided"$]$
$\land$ UNCHANGED $pDecisionValue$

$pAction(p) \triangleq$

network actions independent of the protocol

$\lor\ Deliver(p)$ deliver message

Protocol-specific actions

coordinator has some special actions

$\lor\ \land\ p = Coordinator$

enabled if haven't proposed yet

$\land\ pProtocolState[Coordinator] = \text{“awaitingProposal”}$

$\land\ Proposal(Coordinator)$

$\lor\ \land\ pProtocolState[p] = \text{“seenProposal”}$

$\land\ Ack(p)$

$ProcessAction\ \triangleq$

All possible actions

choose a process to take action

$\exists\, p \in Processes :\ \land \neg pIsCrashed[p]$ enabled if p is not crashed

$\land\ pAction(p)$

$\land$ UNCHANGED $\langle pIsCrashed,\ Coordinator,\ pInitialValue\rangle$

MAIN SPECIFICATION

$Validity\ \triangleq$

All decision values were the initial value of some process

$\forall\, i \in Processes :\ \lor\ pIsCrashed[i] \land pDecisionValue[i] = \text{“”}$ crashed undecided

$\lor\ \exists\, j \in Processes : pDecisionValue[i] = pInitialValue[j]$

$Agreement\ \triangleq$

All decision values are the same

$\forall\, i,\ j \in Processes :\ \lor\ pIsCrashed[i] \land pDecisionValue[i] = \text{“”}$ crashed undecided

$\lor\ pIsCrashed[j] \land pDecisionValue[j] = \text{“”}$ crashed undecided

$\lor\ pDecisionValue[i] = pDecisionValue[j]$

$Liveness\ \triangleq\ \Diamond(Validity \land Agreement)$ should this be $\Diamond\Box$?

$Next\ \triangleq\ \lor\ ProcessAction$

a process can crash at any time between actions

$\lor\ \exists\, p \in Processes :\ \land \neg pIsCrashed[p]$ enabled if p is not already crashed

$\land\ Crash(p)$

$\land$ UNCHANGED $\langle Coordinator,\ pInitialValue\rangle$ these never change

$Fairness\ \triangleq\ \mathrm{WF}_{\langle State\rangle}(ProcessAction)$

$Spec\ \triangleq\ Init \land \Box[Next]_{\langle State\rangle} \land Fairness$

THEOREM $Spec \Rightarrow TypeOK \land Liveness$

B Asynchronous consensus with failure TLC configuration file sample

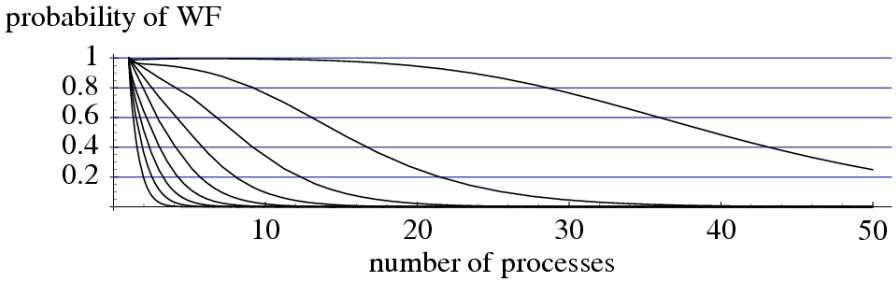
```
SPECIFICATION Spec
PROPERTY Liveness
INVARIANT TypeOK

CONSTANT Processes = p0, p1, p2, p3, p4
CONSTANT Domain = attack, retreat

SYMMETRY Perms
```

C Weak failure detector probability analysis

Chandra and Toueg [1] show that the weakest failure detector that can solve the consensus problem must eventually satisfy the weak accuracy condition. Weak accuracy requires that there is one correct process that is suspected by no other process. It seems that this is actually a fairly difficult requirement to meet in large systems. If we assume a system with n processes each of which has a constant probability ρ of falsely suspecting that a process i has failed, the probability that the failure detectors will satisfy the weak accuracy condition is
$P(n, \rho) = 1 - (1 - (1 - \rho)^{n-1})^n$.



For each of the lines in the above figure from left to right $\rho = 0.9, 0.8 \ldots 0.2, 0.1$. There is an exponential decay in the probability as the system increases in size.

However, it is important to note that in real systems there may be strong correlations between false suspicions. For example, if timeouts are used as a basis for detecting failures then a relatively slow process will tend to be suspected by multiple processes. Thus, the above probability model may not be realistic.

It seems that under the same assumptions, a weaker accuracy condition requiring that some process is *transitively* trusted by every process would not exhibit this exponential decay in probability. It also seems that a failure detector satisfying this condition would suffice for solving consensus (since all processes can receive data from the universally-transitively-trusted process).

References

[1] T. D. Chandra, V. Hadzilacos, and S. Toueg. The Weakest Failure Detector for Solving Consensus. *Journal of ACM*, 43(4):685–722, 1996.

[2] T. D. Chandra and S. Toueg. Unreliable Failure Detectors for Asynchronous Systems. In *10th ACM Symposium on Principles of Distributed Computing*, 325–340, August 1991.

[3] M. Fischer, N. Lynch, and M. Paterson. Impossibility of Distributed Consensus with One Faulty Process. *Journal of ACM*, 32:374–382, April 1985.

[4] R. Geurraoui and A. Schiper. Consensus: the Big Misunderstanding. In *6th IEEE Workshop on Future Trends in Distributed Computing Systems*, 183–188, October 1997.

[5] L. Lamport. Specifying Systems. Boston: Addison-Wesley Longman Publishing Co., 2002.

[6] L. Lamport, R. Shostak, and M. Pease. The Byzantine Generals Problem. *ACM Transactions on Programming Languages and Systems*, 4(3):382–401, July 1982.

[7] N. Lynch. Distributed Algorithms. San Francisco: Morgan Kaufmann Publishers, 1996.