Emulation of a Fault-Tolerant Variant of Safra's Termination Detection Algorithm

Georgios Karlos
Vrije Universiteit Amsterdam
g.karlos@vu.nl

We implemented our fault-tolerant variant of Safra's termination detection algorithm as well as the improved failure-sensitive version of Safra's algorithm in Java. To analyze the functional correctness of these algorithms, emulations were performed on the DAS-4 wide-area distributed system. In all test runs termination was detected correctly by both algorithms. An important motivation for emulating the activity of the basic distributed algorithm is that we could test the functional correctness and performance under a wide variety of different execution scenarios, including crashes of nodes.

The emulation works as follows. Each node runs on its own thread; no information is shared between these threads. Once activated, a node chooses k activities to perform, with $k \leq 4$, and under the uniform or Gaussian distribution (with $\mu = 2$ and $\sigma = 1$). Each activity consists of either some internal events or sending out a basic message. An execution of a basic algorithm on a node is emulated by letting the node's thread sleep for some random amount t_s of ms with t_s in $[0, 2000]$, under the uniform or Gaussian distribution (with $\mu = 1000$ and $\sigma = 200$). To send a basic message, a random node is chosen uniformly to receive that message. Basic messages reach their receiving nodes by calling their RBM procedure after some random bounded delay l_m. This simulates network latency and is always under the Gaussian distribution (with $\mu = 60$ and $\sigma = 10$ ms). Forwarding the token corresponds to an invocation of the RT procedure of the receiver after some random delay l_t. This delay is calculated in exactly the same manner as l_m and is under the same distribution as l_m. All message sending is handled by a `Network` object that runs on a separate thread, so that node-runner threads are not delayed. Another dedicated thread decides beforehand which nodes will be crashed and crashes them throughout an execution run with some probability. Each node has the same probability to crash. Termination may occur while not all chosen-to-crash nodes have crashed yet. For every crashed node a new thread is spawned that informs other threads (nodes) of the crash. This simulates the behavior of the failure detector.

We tested our fault-tolerant version of Safra's algorithm on networks of 16, 48 and 144 nodes. We chose to emulate a decentralized basic algorithm, having $N/2$ nodes initially active for each network size N. For each version, network size and probability distribution we ran a test for each of the following intervals of percentages of crashing nodes: $[0, 0]$ (i.e., a crash-free network), $(0, 20]$, $[20, 40]$, $[40, 60]$, $[60, 80]$, and $[80, 100)$. For each test we performed 1000 runs for an identical initial configuration, resulting in a total of 42000 runs[1]. As stated before, in all runs, the algorithm detected termination correctly. We moreover used the emulation results to compare the performance of the failure-sensitive and the fault-tolerant versions of Safra's algorithm. Clearly these results have to be taken with a grain of salt, as for instance in practice workloads do not always follow a smooth probability distribution and the chosen multi-threaded scheduling policies are likely to introduce some kind of bias with regard to the performance results. (Testing termination detection algorithms on top of specific basic distributed algorithms and a specific hardware platform of course also comes with a certain bias.) Still the presented synthetic results at least give some indication on which overhead the termination detection algorithms may impose and which performance they may exhibit.

In most test runs, we let our emulated basic algorithm produce relatively little activity, with actual termination occurring within 2 to 4 token rounds. This was a conscious choice, because due to the randomization, even a small increase in per-node activity resulted in enough global activity to let a run last for a long time or not terminate at all. This Achilles' heel of our emulation approach is also the reason why we restricted the experiments to networks of up to 144 nodes; runs on significantly larger networks refused to terminate even if the basic algorithm exhibited very little activity. For our main aim, to test functional correctness, this is not such a serious issue, but testing the scalability of our algorithm would require test runs on much larger networks. A possible solution would be to mix randomization with forced termination of the basic algorithm on individual nodes in the long run.

We refer to the failure-sensitive version as FSS and to the fault-tolerant version as FTS. We consider the following metrics to measure the overhead of our algorithm. Each of them is determined per test by taking the means over the 1000 runs in the test.

- MTC (Mean Token Count): the total number of tokens sent.
- $METC$ (Mean Extra Token Count): the number of tokens sent after actual termination.
- $MBTC$ (Mean Backup Token Count): the number of backup tokens sent. (This measure is only relevant to FTS.)

[1] 2 probability distributions, 3 network sizes, and 6 intervals for our fault-tolerant algorithm. Only 1 interval ($[0, 0]$) for the failure-sensitive version.

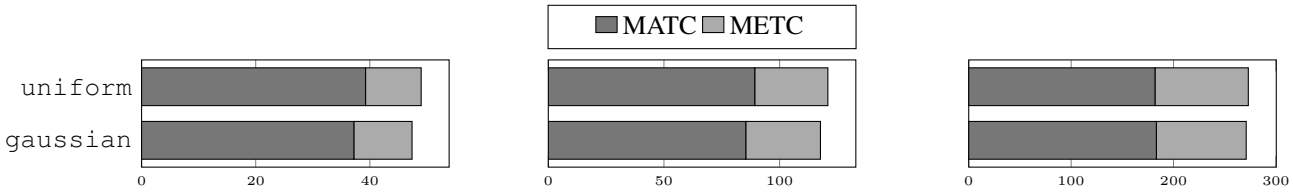

Fig. 1: $MATC_N(\text{FSS}_d^0)$, $METC_N(\text{FSS}_d^0)$ on crash-free networks, with $N = 16$ **(left)**, $N = 48$ **(middle)**, and $N = 144$ **(right)**, where either $d = u$ (uniform) or $d = g$ (Gaussian).

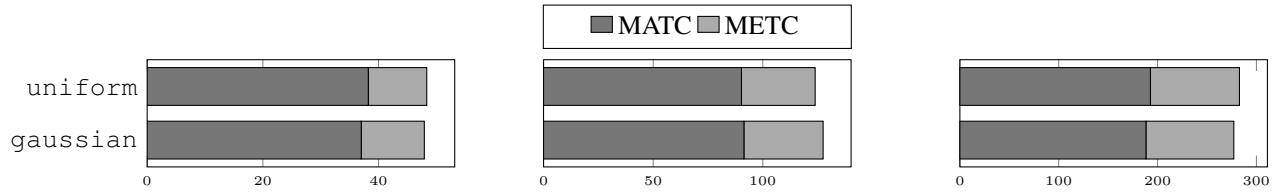

Fig. 2: $MATC_N(\text{FTS}_d^0)$, $METC_N(\text{FTS}_d^0)$ on crash-free networks, with $N = 16$ **(left)**, $N = 48$ **(middle)**, and $N = 144$ **(right)**, where either $d = u$ (uniform) or $d = g$ (Gaussian).

- MBC (Mean Bit Complexity): the average size of the token in bits.
- MPT (Mean Processing Time): the average time it takes to execute an algorithm procedure. Calculated by dividing the total time spent executing algorithm procedures, by the number of times they were executed.

$METC$ can be viewed as an abstract representation of the time it takes for the algorithm to detect termination after termination has actually occurred. MPT is used as an indication of how long the basic algorithm may get delayed, in terms of CPU time, by the termination detection algorithm (e.g. when processing the token).

We use $M_N(V_d^c)$ to denote the metric M of version V running on a network of size N under distribution d and with $c\%$ of the nodes chosen to crash (e.g. $MTC_{16}(FTS_g^0)$ denotes the value of MTC on a crash-free network of 16 nodes, under the Gaussian distribution).

We compared the performance of FSS and FTS on the three aforementioned network sizes, first without crash failures. According to Figures 1 and 2, FTS on average sends no more control messages than FSS, and requires the same number of control messages to detect termination after it has occurred. In that figure, $MATC$ (Mean Actual Token Count) refers to the number of tokens sent before termination occurred, i.e., $MTC - METC$. It indicates that our fault-tolerant version imposes no additional control message overhead, in the absence of crashes. Furthermore, for both FSS and FTS, $METC_N \leq N$ for all N, meaning that termination is detected, on average, within one round after it has actually occurred. There is no significant variation between the two distributions.

We also experimented with the original version of Safra's algorithm, which was found to require significantly more control messages than the improved version to detect termination after it has occurred. These experimental results are not included here, because our focus is on fault tolerance.

Figure 3 summarizes the results on crash-prone networks. FTS produces a larger number of tokens in crash-prone networks than in crash-free ones. This is due to the fact that in a significant number of test runs, upon arrival of the token t at a node i, $\text{REPORT}_i \setminus \text{CRASHED}_t \neq \emptyset$. In such cases, where the token is propagated faster than that local failure detectors detect a crash, line 22 of HT_i (Procedure 9) will produce an additional round. Once this has happened though, the rest of the alive nodes will also detect the crash through their failure detectors within that extra round.

FTS performed as expected with respect to the number of backup tokens issued. However, $MBTC$ exhibits a disproportionately small increase when the network size N grows. For example, $MBTC_N$ roughly doubles going from the small to the medium network, whereas N increases by a factor of 3. A backup token is issued only from the predecessor of the crashed node and only if that node learned of the crash through its failure detector. Thus, with more nodes crashing, more backup tokens are issued. However, the rate of this increase in $MBTC_N$ is lowered. For example, on the medium network, with 20% of the network crashing (9 nodes), there are about 4 backup tokens, whereas when 80% of the network crashes (38 nodes), this number is only (roughly) doubled. This, and the fact that the number of issued backup tokens is generally observed to be low, indicate that in our emulations, crash knowledge is mostly propagated by the token. There is no significant variation between the two distributions.

MBC was measured by the size of the Java object representing the token. We note that our implementation follows strictly the algorithmic description and thus does not include possible optimizations available in the context of crafting actual messages. For example, in our implementation N integers are reserved for $count_t$, whereas when crafting an actual message we could optimize by using space only for positive values. This is more apparent when comparing MBC between FSS and FTS with no nodes crashing. Nevertheless, the measurements still provide some meaningful information,

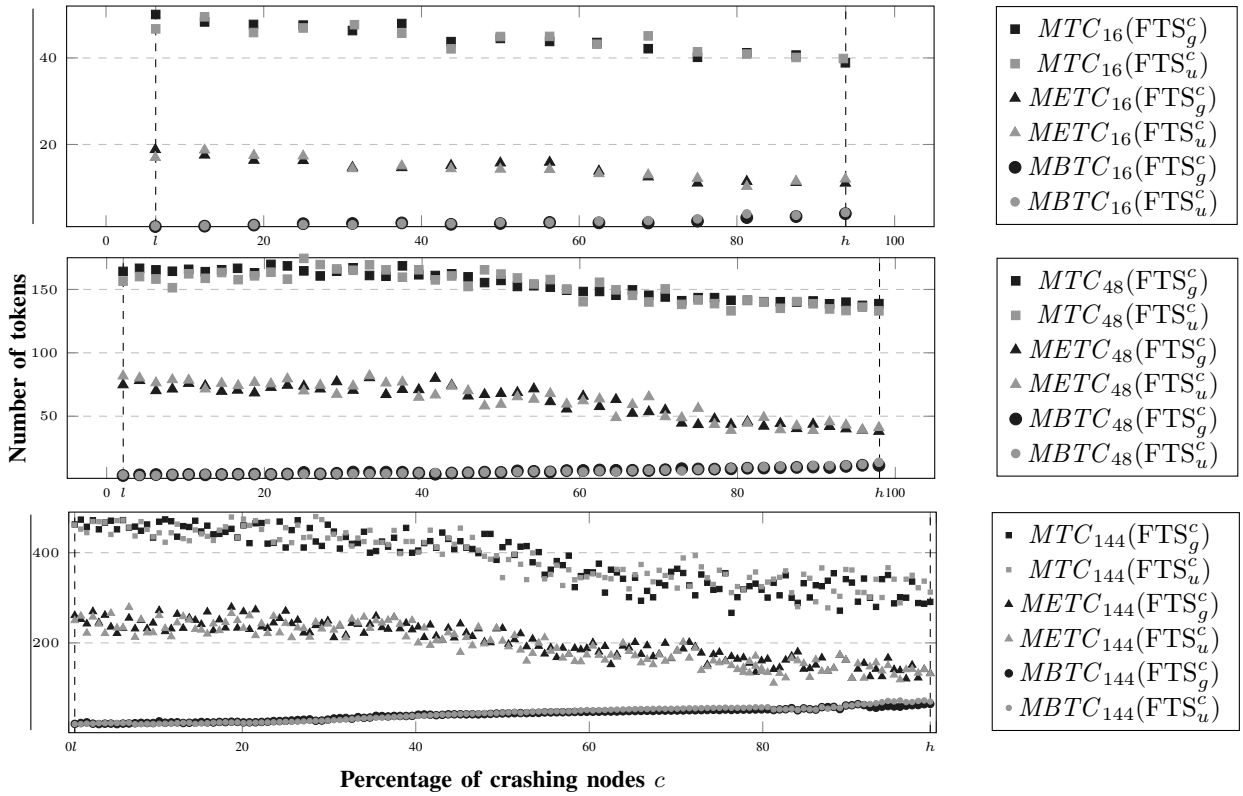

Fig. 3: MTC_N, $METC_N$, $MBTC_N$ of FTS^c_d on crash-prone networks with $N = 16$ **(top),** $N = 48$ **(middle),** $N = 144$ **(bottom)**, with c the percentage of crashing processes, $\frac{c}{100} * N \in [l, h]$ and $d \in \{u, g\}$.

——		**Crashing:**	**0-0%**	**0-20%**	**20-40%**	**40-60%**	**60-80%**	**80-100%**
MBC_{16}(FSS)	889	MBC^{16}(FTS)	2272	2564	2718	2822	2902	3012
MBC_{48}(FSS)	872	MBC_{48}(FTS)	3798	4082	4564	5004	5228	5448
MBC_{144}(FSS)	862	MBC_{144}(FTS)	6844	7258	8700	9585	9723	9988

Fig. 4: MBC_N(FSS) and MBC_N(FTS) in bits, on networks of size N for different values of N. The average of the two distributions is taken.

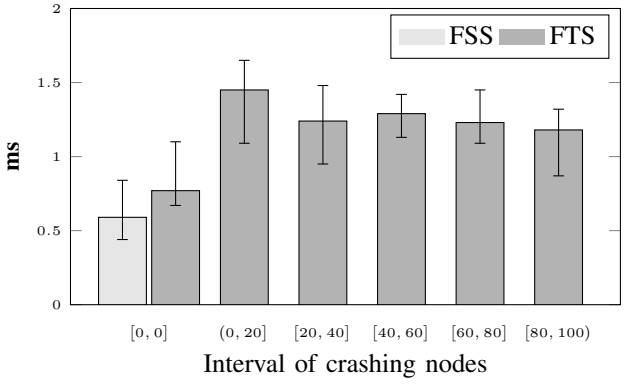

Fig. 5: MPT(FSS) and MPT^c(FTS) for different c intervals. Average over the three networks and the two distributions.

which is summarized in Figure 4. As expected, MBC_N(FSS) is more or less stable, while $MBC_N(FTS)$ increases with the network size due to the $O(N)$ bit complexity of FTS. Furthermore, it increases as the number of crashing nodes increases, which can be attributed to the fact that $\textsc{Crashed}_t$ temporarily grows in size when a node crashes.

Figure 5 summarizes the mean processing time at a node required by FTS compared to FSS. Error bars depict the standard deviation. As expected, FTS is slower on crash-free networks, due the added concurrency per node. The difference is rather small, because when no failures occur, there are relatively few updates on shared data structures. In case of crashes, for FTS the processing time at nodes roughly doubles, compared to crash-free networks. In case of many token rounds, instead of only 2 to 4 in the emulations, this difference would actually decrease significantly, because there would be many crash-free rounds. The fact that the processing time stays stable when the number of crashes increases is also due to the small number of token rounds, because in our emulations the token tends to carry information on multiple crashes.