1. General overview of the architecture

The software is a simulation of a distributed fault tolerant system for summing up integers. The general architectural approach is a master – slave one. Each master is responsible for starting a set of slave nodes and assigning them work. When all the work available is assigned to the slaves, the master requests the results that the slaves have produced. Commission failures are simulated. An approach that mitigates the risk of receiving a faulty result is implemented by duplicating the computation on a number of separate nodes and comparing the output results. For this purpose, logic for agreement is implemented. The general overview o f the architecture could be seen in Figure 1.

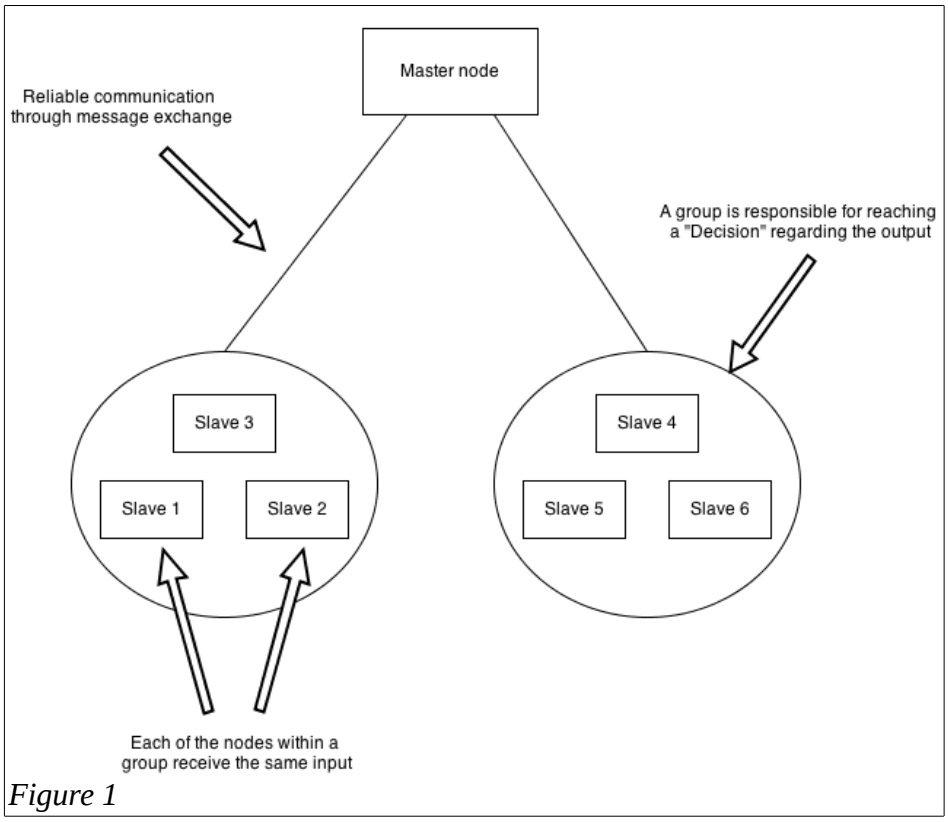

Figure 1

2. Communication paradigm

The master and slave communicate through the exchange of messages of a predefined maximum size. The communication is done via a central Network component that is responsible for routing the messages to the correct destination. In order to make the scenario more realistic, this component can randomly omit the delivery of a message. Each time a message is sent, the network component tosses a biased coin with predefined probability bias and based on that decides whether the message should be delivered or not. This renders part of the messages sent from master to slave and the other way around omitted. In order to ensure against dropped messages causing incomplete computations , a reliable transport

protocol is implemented on the top of the network component. This is done though the creating of individual sockets for each connection. Those sockets maintain the state of the connection and make use of a stop and wait algorithm in order to ensure reliable delivery. Re transmission timers are also used in order to decide how long is the appropriate period waiting for an ACK before re-transmission. If a single message is re-transmitted more than N times (where N is some predefined number), the node is labeled unreachable and excluded from the list of nodes that the master uses. In order to ensure against duplicate messages (due to lost ACKs), sequence numbers are used. In Figure 2, one can see some of the communication scenarios that are possible. In order to simulate realistic conditions, the communication between the master and the slave nodes is performed via a Network component.

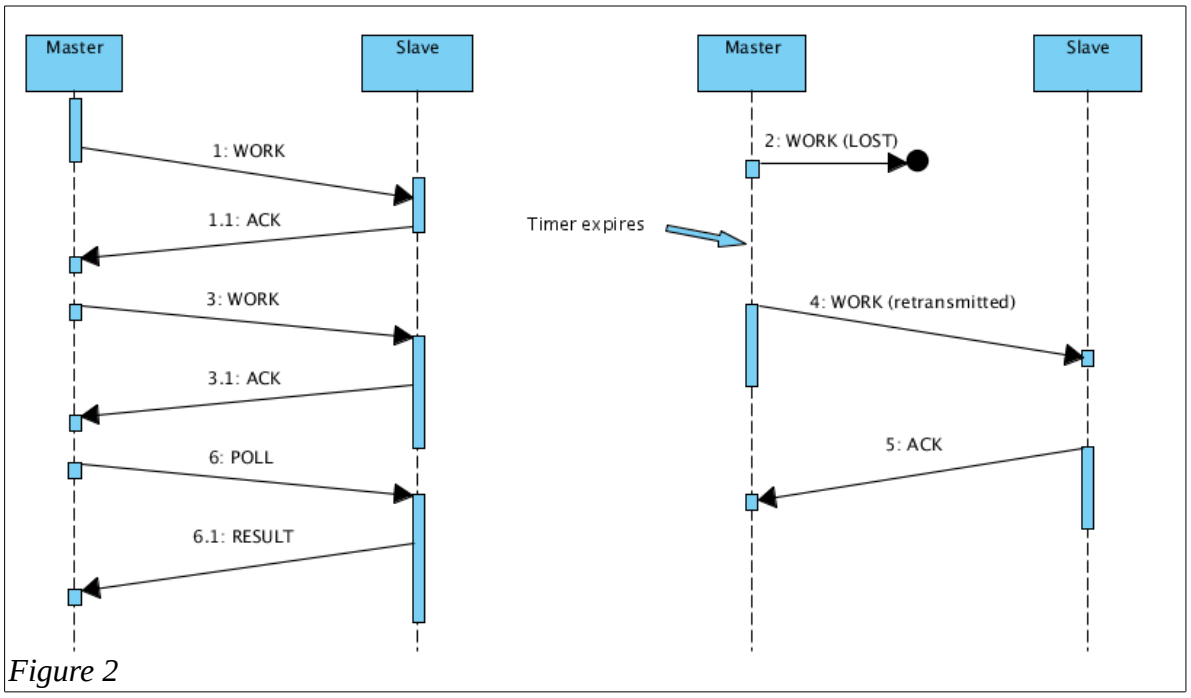

Figure 2

The diagram shows what happens in a successful scenario compared to a scenario in which one of the work messages are lost. But those are not all the possibilities. For example it might be that the initial WORK message is delivered but its ACK gets lost. In this case the master's socket times out and re-transmits the message. Since the readmitted message has a different sequence number than the one specified by the slave, it is ignored and ACKed again. For simplicity sake, the protocol does not provide pipe-lining through the use of a sliding window mechanism. There are several types of messages used in the system;

- **WORK** – this is a message containing an array of integers that is sent by the master to the slave. Upon receipt of this message, the master sums up its content and adds it to its partial sum. After that the message is ACKed by sending an ACK message.

- **ACK** – this type of control message is used to acknowledge the successful reception of a WORK message.

- **POLL** – when the master needs to request the result of a computation from the slave it

sends a POLL message.

- **RESULT** – when the slave receives a POLL message, it sends a result message containing the result of the computation performed up to the moment. It is important to note that the result message serves as an implicit ACK for a POLL message.

3. Tolerance towards byzantine failures

In order to simulate commission failures, upon delivery of a result each node throws a biased coin in order to determine whether to come up with a random result or to deliver the correct one. In order to prevent the final result of the computation from the effect of this simulated problem, each node can be replicated a number of times. This is done by constructing groups of nodes that the master has knowledge of. The master collects the results of all nodes that are withing a group and decides whether an agreement on a result can be reached. The rule used for that is a simple majority function that requires 50% + 1 nodes to have the same result in order to reach an agreement. Upon receiving a result message, the master assigns the result to the correct group (depending on the source of the message) and checks whether a group had reached an agreement. A group can be in three different states:

- **DECIDED** – the group can transition into a decided state at any time after 50% + 1 matching results are gathered. It is not needed for all the nodes to return their result in order for the group to transition into this state. For example the following log of a group state demonstrate a scenario

```
Members [SLAVE-70 SLAVE-69 SLAVE-68 SLAVE-65 SLAVE-67 SLAVE-66 SLAVE-72 SLAVE-71 ] Results [ {R: 53400 O: 5} ] State [DECIDED]
```

  in which a decision is reached before all the slave nodes returned their results. In this scenario the group has 7 nodes each working on the same input. Five of them have already returned the result 53400. Since 5 nodes are more than 50% + 1, the group has transitioned in a DECIDED state without waiting for the other nodes.

- **UNDECIDED** – In this state the group does not have enough data to reach an agreement. For example the following trace shows a scenario in which only one result have been delivered. Therefore there is no way to decide on a result.

```
Members [SLAVE-70 SLAVE-69 SLAVE-68 SLAVE-65 SLAVE-67 SLAVE-66 SLAVE-72 SLAVE-71 ] Results [ {R: 53400 O: 1} ] State [UNDECIDED]
```

1. **FAILED** – If a group transitions into this state, there is no way to recover and the whole computation cannot be finished. This situation can occur in several different cases. For example If the group has a number of different results and this number is larger than 50%+1, there is not way to reach an agreement no matter if the other nodes return the same result. Furthermore, if a significant number of nodes have dropped from the system (due to reaching the re-transmission timeout), and are dis-joined from the group, a result cannot be agreed upon. For example if only one node is left in the group (where initially they have been 7), then it is impossible to achieve the number of results needed for agreement, simply because there are no nodes to deliver them. In the case of a group agreement failure, the whole computation is interrupted and needs to be restarted. For example in the trace below , there is no way to reach an agreement , no matter how many more messages are received.

```
Decision cannot be reached...Members [SLAVE-9 SLAVE-14 SLAVE-13 SLAVE-16 SLAVE-15 SLAVE-10 SLAVE-12 SLAVE-
11 ] Results [ {R: 58617 O: 4}  {R: 2 O: 1}  {R: 80 O: 1}  {R: 58 O: 1}  {R: 89 O: 1} ] State [FAILED]
```

4. Tests performed

When implementing the system, various tests were performed in order to observe its behavior. For example, setting the probability of dropping packets to a significant value such as 30 %, we can observe that a lot of nodes are dis-joined from groups and some of the groups transition into FAILED state due to node insufficiency. This FAILED state can be achieved also by setting the probability of commission error too high. This produces too many different results per group and agreement cannot be reached. Generally setting the two probabilities below 10 percent, is a good way to observe how the system copes with errors and adapts in order to produce a correct result.

Running the application.
The application can be run through the ant build file provided. The target to be used is “run”. In addition, a simple user interface is provided along with the system. The interface allows for specifying parameters and observing the behavior of the system through the console output window. The interface can be run through the StartGUIMode class.

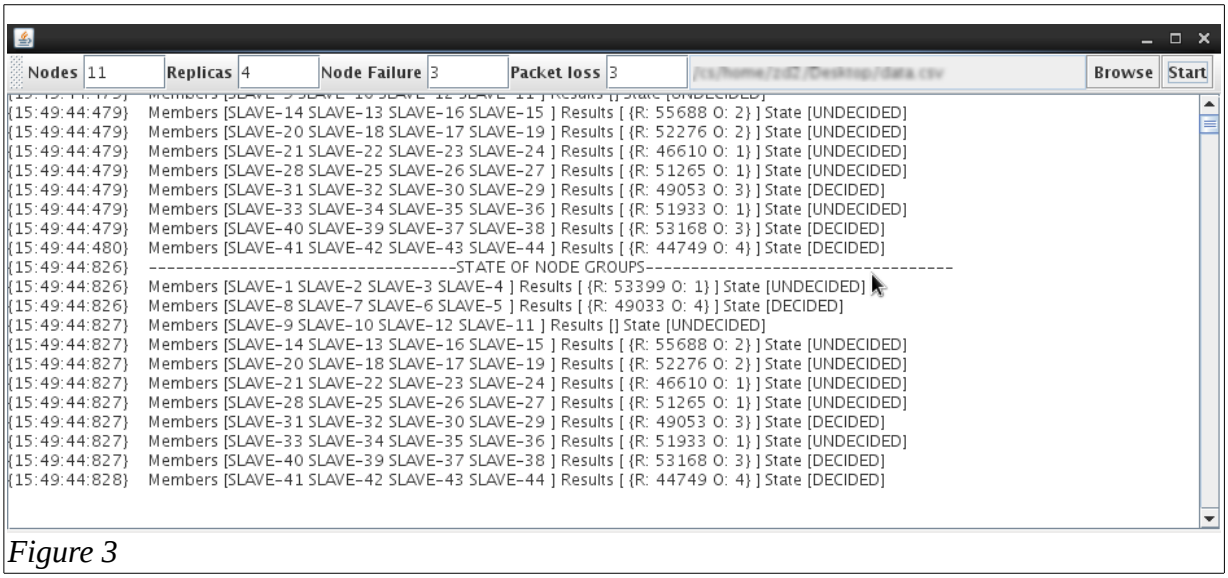

Figure 3

References:

Bessani, Alysson N et al. “Making Hadoop MapReduce Byzantine Fault-Tolerant.” *Work* 49.4 (2010) : 1189-1204.

Costa, Pedro et al. “Byzantine Fault-Tolerant MapReduce: Faults are Not Just Crashes.” *2011 IEEE Third International Conference on Cloud Computing Technology and Science* 2011 32-39.

Lamport, Leslie, Robert Shostak, and Marshall Pease. “The Byzantine Generals Problem.” *ACM Transactions on Programming Languages and Systems* 4.3 (1982) : 382-401.