Implementing fault tolerant agreement in JaCaMo

Endre Fülöp[1]

Eötvös Loránd University, Budapest, Hungary

Abstract. Implementing algorithms in multi-agent frameworks can be challenging, because the special requirements of distributed systems meet with the incomplete information, openness assumption, and social issues, all of which are present in certain multi-agent systems. The issue is further aggravated by the fact that the software engineering environment did not have as much time to mature as it had in case of other programming paradigms. In this article an implementation of agreement is given which is based on the Byzantine Fault Tolerance algoritm, and its improvement which adapts the original one to asynchronous environments. The implementation also takes reusability into account by providing a source code library which provides an API for its consumers. After detailing the key aspects of the implementation, software engineering aspects, security and practical considerations are analyzed.

Keywords: Multi-agent systems, Agreement · Fault tolerant behaviour · Social interaction

1 Acknowlegement

This work was supported by the European Union, co-financed by the European Social Fund (EFOP-3.6.3-VEKOP-16-2017-00002).

2 Fault tolerant agreement

The motivation behind this implementation is the open, and hetegeneous nature of certain multi-agent systems, where the individual agents are not necessarily under control of a single authority[4] In such systems reaching total agreement is a desirable goal, however sometimes it may not be feasible. By reducing the scope of the agents, who come to an agreement at the end of the agreement process, some degree of coordinated behaviour can be achieved.

2.1 Byzantine behaviour

The Byzantine Generals Problem[3] is well studied field of distributed computing, and in its original form it gives necessary requirements for replicating data between distributed nodes of computation in a fault tolerant way. The group of

distributed systems problems where there are processors that send inconsistent information to other processors, thus appearing to be working for some nodes, and faulty for some others, are called byzantine, becase of their resemblence to the Byzantine Generals problem. The implementation is based on the algorithm introduced in Lamports original paper, but takes the newer practical algorithm [2] into account, so that it is more fitting for the inherently asynchronous nature of the multi-agent framework environment.

2.2 Byzantine Generals Problem

The Byzantine Generals Problem is the following: there are generals surrounding the enemy city. They are in such distance away from each other, that they can only communicate by sending messengers. The battle can only be won if enough of the generals agree on the same course of action. If all generals were loyal the the agreement would be trivial, however there are some traitors amongs them, who want to sabotage the attack. The base problem assumes a perfect communication channel between every participant. It is shown in the paper, that with simple oral messages if the number of traitors is less then one third of all generals, then there is an algorithm which guarantees, that at the end of the process, all the loyal generals agree upon the same plan of action, and this small number of traitorous generals cannot misguide them. The algorithm is initiated by a particular general, the commanding general. It is permitted for the commanding general to be traitor, and the algoritm has the same guarantees. The algorithm can further be improved by using cryptographically signed messages, and its applications in not fully-connected communication graphs is further examined.

2.3 Practical Byzantine Fault Tolerance

The original Byzantine Fault Tolerance[3] have requirements, which make it impractical to apply in a setting, where communication is performed in an asynchronous manner. The timing estimates of receiving a message, and the decideability of the fact that whether a general sent a message or not is tied together, and therefore the algoritm is implicitly synchronous. The pBFT [2] algrithm however does not make any assumptions about the timing of the communication, and its liveness properties can be shown in case an uncooperative or malfunctioning participant can delay messages of even correct one, but not indefinitely. The main idea behing the algorithm is that after the correct amount of messages is sent, the agreement between the correct participants eventually emerges. So each node in the network waits for a predefined amount of the same choice before accepting it as the result of the agreement. The paper further analyzes optimizations of the algorithm, however the implementation here does not concern itself with them, as expressing even the base algorithm in the framework not trivial. In the future, more advanced versions of the algorithm can be examined in the context of multi-agent systems as well.

3 Implementation

3.1 Implementation with agents

The original and also the practical Byzantine Fault Tolerance algoritms were created to solve a fault tolerance problem in a non-autonomous system. However in multi-agent systems agents are autonomous, and not only faults can hinder an agreement attempt, but agents themselves can choose not to cooperate if their goals and desires dictate. Based on the requirements of the original versions of the algoritms, the agreement of the majority of the agents can be guaranteed, if the number of non-cooperative agents is less then one third of the agents total amount. Non-cooperation, however can be deliberate, or can be a malfunction on the agents part. The main motivation to use byzantine algoritms in multi-agent systems is to agree on something common, and they are most applicable when the agents are motivated to come to an agreement.

3.2 Practical Byzantine Agreement implementation

If there are n uncooperative agents, then there must be at least $2n+1$ cooperative agents. If this requirement does not hold, then the algoritm cannot guarantee agreement. Each of them knows that for which fact which other agents they have to agree with. They also know the default value of the agreement, should the agreement process fail. The supposed treshold of traitors is also known by them, however this one is optional, as all of the agents could very well use the lowest possible value which can be computed from the size of the agreement group. The commanding agent is one of the $m = 3n + 1$ total agent, who can be chosen randomly. The commanding agent sends its choice everybody else, and its role is finished. Every cooperative agent, when receiving a message with a choice, sends the same choice to every other agent, except from whom the message was received. The cooperative agents also keep track of the different choices they encountered, and their inremental popularity, which is a running total per choice. At the moment a message with choice c would increment its corresponding counter in the belief base of the agent to match the the value of $n + 1$, the agent decides that the agreed choice is c. However it does not stop sending messages until the right amount messages are sent by him. The stop criterion is detailed later. Eventually every cooperative agent should come to the conclusion, that the agreed choice is c.

3.3 The framework

JaCaMo[1] is a framework for developing multi-agent applications. It brings several other frameworks under the same umbrella, unifying their tools, and capabilities, and provides glue between the different components. It uses Jason for agent-orinented, CArtAgO for environment-oriented, and Moise for organization-oriented progragramming, and also provides meaningful tooling for monitoring and debugging some aspects of the multi-agent system under development.

It is natural in the BDI paradigm to represent the factual knowledge of agents separately from the functional knowledge. The functional knowledge is what enables an agent to achieve its goals, and in the Jason framework it is modelled with plans. Plans are declarative sequences of actions that should be taken by an agent responding to an event. There are multiple ways for an event to be emitted. Belief base updates trigger events by default, and these updates can be consequences of the agent percieving the external environment, making mental notes, and also acquiring and failing goals.

The agent-framework makes it possible to model the problem with many tools, however this implementation uses only one such tool, the Jason part of JaCaMo. This is a conscious decision, as expressing how to communicate the intent to come to an agreement is best implemented where it is most idiomatic. The environment part of the framwork not used for this implementation, neither is the organization. This also means that this implementation does not pose any constraints on those parts on any project that would like to reuse the solution, and even the agent part is implemented in a non-intrusive way.

3.4 Agreement library

The implementation is in form of a source code library, which provides the necessary plans that an agent must have in order participate in the agreement process. The plan descriptions can be distributed in a .wsl file, and can be included in other agent's description files via the preprocessor. There are however other requirements for the libarry to work.

The algorithm is ultimately specified by a plan for an achievement goal named `byzantine_agree`, which has two parameters, the topic of the agreement, which can be used to identify the object of agreement at hand, and the choice that is chosen by the commanding general. When an agent choses to pursue this goal, the plan library will contain the necessary implementation to initiate the agreement algoritm. The commanding agent role is identified by having this goal at the beginning. The choice of the commanding general is not crucial in our case, but in order to be able to handle a traitorous commander, there is some necessary context information that must be present at every agent participating in the agreement.

Every participant should know what kind of topics exist, and which agreement group should decide the outcome. Also to provide a convenient API, there is a `agreement_treshold` belief that can be specified. This signifies the believed upper threshold of non-cooperative participants in the group. This parameter could also be inferred from the size of the agreement group, however it allows for less communication to be used if the specific agreement instance requires less redundancy.

Listing 1.1. An example initial state of the belief base

```
agreement_group(topic(default), [a, b, c, d, e]).
agreement_treshold(topic(default), 1).
```

The main entrypoint for the algoritm designates how the commanding agent should start the agreement. It has two parameters, the topic, and the choice of the commander. Because the commander will not receive further messages during the agreement, and the decision of the final choice is given in this implementation in a reactive manner, the result of the agreement is finalized in its belief base. Then the belief base is consulted for relevant targets of communication for the triggering topic. The variable `AG` is unified with a list of agent names, who are belived to part of the agreement group. Then the achievement goal `byzantine_agree_rec` is triggered as a subgoal of the entrypoint.

Listing 1.2. Main entrypoint

```
+!byzantine_agree(TOPIC, FACT)[source(self)] <-
  +final_fact(TOPIC, FACT)
  ?agreement_group(TOPIC, AG)
  !byzantine_agree_rec(TOPIC, FACT, AG, 0).
```

The message sending and verification logic is most densely concentrated in the implementation of the main recursor `byzantine_agree_rec`. The plan has a context named `byzantine_valid_request`, which is a predicate, and decides whether the incoming request should be taken into consideration. This is necessary in order for the implementation to be robust. Some of the precautions taken by this implementation are specific to the theoretical algorithm, but others are given for further hardening.

In case the plan is deemed relevant, the source of the current message is noted by the agent. Then an achievement goal of incrementing an internal counter for the received choice: `byzantine_fact_count_increment`. This goal will also detect if a big-enough count is reached. Then there is a conditional recursive invocation of this goal. This means that the goal is triggered in the targets of the `.send` built-in communication primitive.

Listing 1.3. Main recursor

```
+!byzantine_agree_rec(TOPIC, FACT, AG, STEP)[source(SOURCE)]
  : byzantine_valid_request(TOPIC, AG, STEP, SOURCE) <-

  +agreement_message_for_step(TOPIC, STEP, SOURCE);
  !byzantine_fact_count_increment(TOPIC, FACT);
  ?agreement_treshold(TOPIC, TRESHOLD);
  if (STEP <= TRESHOLD) {
    .my_name(ME);
    .difference(AG, [ME, SOURCE], OTHERS);
    .send(OTHERS, achieve,
      byzantine_agree_rec(TOPIC, FACT, OTHERS, STEP + 1)
    );
  }.
```

It is supposed, that the cooperative agents share the same, or equivalent implementation for this goal, and byzantine behaviour emerges mainly by overriding this plan, sending spurious and false messages.

The validation is implemented as rule in the Jason language, and is designated to check that specific invariant properties hold in each step of the algorithm. One of the invarint properties is that messages should be received only from members of the agreement group, or in the very special case of the commanding agent from `self`. Another invariant is that the targets of the communication should be a valid subset of the initially believed agreement group. The third property defines a variant property, the length of the target agreement group should be decresing with each step. The fourth part of the validator expression is a countermeasure against message flooding, as each participant is allowed to send at most one message to every other participant is each step.

Listing 1.4. Main recursor validation

```
byzantine_valid_request(TOPIC, AG, STEP, SOURCE) :-
  // Consult the belief-base for relevant information.
  agreement_treshold(TOPIC, TRESHOLD)[source(self)] &
  agreement_group(TOPIC, INITIAL_AG)[source(self)] &

  // Only accept messages from members of the initial
  // agreement group.
  (.member(SOURCE, INITIAL_AG) | SOURCE = self) &

  // The target agreement group is a subset of the initial.
  .difference(AG, INITIAL_AG, DIFF) & .empty(DIFF) &

  // The size of the target agreement group is correct with
  // respect to the announced step.
  .length(AG) == .length(INITIAL_AG) - STEP &

  // There is no double vote for this specific topic-step
     pair
  // from the source.
  not agreement_message_for_step(TOPIC, STEP, SOURCE)[source(
     self)].
```

Counting is the facts is done by providing 2 alternative plans, one for the case where the fact was not counted before, and one where the already counted number must be incremented. The check for agreement is done in both cases, because if the treshold is 0, then the agreement can be considered final after the arrival of the first message.

Listing 1.5. Count

```
+!byzantine_fact_count_increment(TOPIC, FACT)[source(self)]
  : byzantine_fact_count(TOPIC, FACT, N)[source(self)] <-

  -+byzantine_fact_count(TOPIC, FACT, N + 1);
  !byzantine_try_settle(TOPIC).


+!byzantine_fact_count_increment(TOPIC, FACT)[source(self)]
```

```
  : not byzantine_fact_count(TOPIC, FACT, _)[source(self)] <-

  +byzantine_fact_count(TOPIC, FACT, 1);
  !byzantine_try_settle(TOPIC).
```

If there is a majority choice with a count greater than the presupposed number of faulty agents, then the agent considers the agreement done, however its responsibility to relay the message is not neglected.

Listing 1.6. Deciding on an agreement

```
+!byzantine_try_settle(TOPIC)[source(self)]
  : agreement_treshold(TOPIC, TRESHOLD)[source(self)] <-

  !byzantine_majority_choice(TOPIC, CHOICE, COUNT);
  if (COUNT > TRESHOLD) {
    +final_fact(TOPIC, CHOICE)
  }.
```

It also interesting to provide an alternative plan for main recursive goal, because if this is triggered, the agent can be sure that at least one non-cooperative or faulty agent exists in the system.

Listing 1.7. Detecting error with alternative plan

```
+!byzantine_agree_rec(TOPIC, FACT, AG, STEP)[source(SOURCE)]
    <-
  .print("Illegal message received: ",
    FACT, " from ", SOURCE, " in step ", STEP, "!"
  ).
```

Listing 1.8. Usage of the libarary in the commander

```
/* Initial beliefs and rules */

agreement_group(topic(default), [a, b, c, d]).
agreement_treshold(topic(default), 1).

/* Initial goals */

+!start <-
  !byzantine_agree(topic(default), choice).

/* Plans */

{ include("inc/agreement.asl") }
```

4 Analysis of noncooperative behaviour

There are multiple ways for an agent to be non-cooperative when participating in an agreement group. The various kinds of behaviours that such agents can produce can be classified into three categories, non-conforming behaviour, detectable misbehaviour, undetectable misbehaviour.

4.1 Non-conforming behaviour

If the agent lacks the basic building blocks of the algoritm, then it is possible that its participation in the agreement algorithm is only nominal. One such example could be, when the agent is not responsive, or does not even have plans for handling goal that are dictated by the agreement process.

Listing 1.9. Example of non-conforming behaviour

```
/* Non-cooperative, passive */
/* +!byzantine_agree_rec(FACT, AG, TRESHOLD, STEP)[source(S)]
     <-
  print("Not doing anything!").
*/
```

4.2 Detectable misbehaviour

Every agent is classified to misbehave in a detectable manner, if its messages trigger the alternative plan of the main recursive goal `byzantine_agree_rec`. This practically means that agents who are active, but manipulate the the target agreement group, the value of current step for an ongoing agreement cause the trigger of the alternative plan.

Listing 1.10. Example of detectable misbehaviour

```
/* Non-cooperative, does not increment step */
+!byzantine_agree_rec(TOPIC, FACT, AG, STEP)[source(S)]
  : byzantine_valid_request(TOPIC, AG, STEP, SOURCE) <-

  +agreement_message_for_step(TOPIC, STEP, SOURCE);
  !byzantine_fact_count_increment(TOPIC, FACT);
  ?agreement_treshold(TOPIC, TRESHOLD);
  if (STEP <= TRESHOLD) {
    .my_name(ME);
    .difference(AG, [ME, SOURCE], OTHERS);
    .send(OTHERS, achieve,
      byzantine_agree_rec(TOPIC, FACT, OTHERS, STEP)
    );
  }.
```

4.3 Undetectable misbehaviour

There are ways for a misbehaving agent to participate in the agreement process and appear to be fully cooperative for some subset of other participants. The key difference here is that the agent does not try to manipulate the structure of the agreement algoritm, it only tries to misinform the other participants with incorrect declaration of which choice it received, or which turn it is currently voting for.

Listing 1.11. Example of undetectable misbehaviour

```
/* Non-cooperative, byzantine */
+!byzantine_agree_rec(TOPIC, FACT, AG, STEP)[source(S)] :
  : byzantine_valid_request(TOPIC, AG, STEP, SOURCE) <-

  +agreement_message_for_step(TOPIC, STEP, S);
  !byzantine_fact_count_increment(TOPIC, FACT);
  if (STEP <= TRESHOLD) {
    .my_name(ME);
    .difference(AG, [ME, S], OTHERS);
    for ( .member(X, AG) ) {
      .random(R)
      .if (R < 0.5) {
        .send(X, achieve, byzantine_agree_rec(TOPIC,
          malicious_fact(targeted_for(X)), OTHERS, STEP + 1))
            ;
      } else {
         .send(X, achieve, byzantine_agree_rec(TOPIC,
         FACT, OTHERS, STEP + 1));
      }
    }
  }.
```

5 Conclusion

Byzantine algoritms are mainly used in distributed systems to provide a solution for fault tolerance with as little redundancy as possible. Multi-agent systems are also distributed in the sense, that not only the execution of logic, but also the decision making is decentralized, and therefore byzantine algrithms can provide benefits for lower layers of implementation, as well as the upper levels, where the cognitive modelling of autonomous behaviour happens. There are many more advanced exmaples of implementing byzantine algoritms in traditional systems, however in multi-agent systems the special characteristics of software engieneering solutions, like frameworks and languages make the implementation not as straigtforward. The JaCaMo framework is very capable at providing a sandbox, where experimental multi-agent solutions can be studied. As further improvement, based on the experience gained, more modern byzantine algoritms could be considered, where the exponential amount of messages could be reduced already mentioned in the pBFT paper. Another valuable extension would be, to use the organization-oriented aspects of JaCaMo to provide an automated commander selection solution.

References

1. Boissier, O., Bordini, R.H., Hübner, J.F., Ricci, A., Santi, A.: Multi-agent oriented programming with jacamo. Science of Computer Programming **78**(6), 747–761 (2013)

2. Castro, M., Liskov, B., et al.: Practical byzantine fault tolerance. In: OSDI. vol. 99, pp. 173–186 (1999)
3. Lamport, L., Shostak, R., Pease, M.: The byzantine generals problem. ACM Transactions on Programming Languages and Systems (TOPLAS) **4**(3), 382–401 (1982)
4. Wooldridge, M.: An introduction to multiagent systems. John Wiley & Sons (2009)