Pre-Proceedings of 8th International Workshop on Engineering Multi-Agent Systems
(EMAS 2020 @ AAMAS), 8-9 May 2020, Auckland, New Zealand.
Eds. Cristina Baroglio, Jomi F. Hübner, and Michael Winikoff.

Fault Tolerance in Multiagent Systems

Samuel H. Christie V and Amit K. Chopra

Lancaster University, Bailrigg, Lancaster, LA1 4YW, UK
{samuel.christie,amit.chopra}@lancaster.ac.uk

Abstract. A decentralized multiagent systems (MAS) is comprised of autonomous agents who interact with each other via asynchronous messaging. A protocol specifies a MAS by specifying the constraints on messaging between agents. Agents enact protocols by applying their own internal decision making.
Various kinds of faults may occur when enacting a protocol. For example, messages may be lost, duplicates may be delivered, and agents may crash during the processing of a message. Our contribution in this paper is demonstrating how information protocols support rich fault tolerance mechanisms, and in a manner that is unanticipated by alternative approaches for engineering decentralized MAS.

1 Introduction

Like any software system, a multiagent system is vulnerable to a variety of faults resulting from any number of root causes: bugs, hardware failure, environmental conditions, etc. If handled poorly or not at all, such faults could propagate through the system and ultimately cause an *error*, or deviation from the specified behavior of the system.

This paper is concerned with decentralized multiagent systems (MAS) in which autonomous agents communicate via asynchronous messaging and coordinate their computations by following an interaction protocol. The decision making of agents being private, the protocol is in fact the fundamental operational specification of a MAS. Indeed, it is meaningless to talk about the computations of a MAS except in terms of messages sent and received by its agents.

Agents enact a protocol by plugging in their private decision making, as encoded in their policies. Although there has been significant work on protocol specification [3, 10] and engineering protocol-conformant agents [1], there is little work that addresses protocol enactment under various kinds of faults. The faults may correspond to communication infrastructure failures, e.g., message loss, corruption, duplication, and so on, or to agent failures, e.g., crashes.

We would expect that a fault-tolerant MAS has the following two properties. One, no fault causes an agent to send a message that would be noncompliant with the protocol. We refer to this property as *compliant-despite-faults*. Two, agents, if they choose to, can recover from faults by sending additional messages. We refer

Listing 1. The Ridesharing *RFQ* Protocol

```
RFQ {
  role Rider, Service
  parameter out ID key, out location, out destination, out price

  Rider -> Service: Request[out ID key, out location, out destination]
  Service -> Rider: Offer[in ID key, in location, in destination, out price]
}
```

to this property as *progress-despite-faults*. Naturally, any additional message must be protocol-compliant.

Our contribution in this paper is illustrating how fault-tolerant MAS with the above properties can be constructed. We illustrate our ideas in the framework of information protocols [9]. In fact, we show that whereas information protocols are naturally compatible with fault-tolerance mechanisms, alternative approaches are not. In particular, although alternatives can ensure compliance, progress despite faults would be challenging.

2 Basic Fault Handling with Information Protocols

Throughout this paper, we will be developing examples based on a hypothetical ridesharing service.

Listing 1 gives the BSPL specification for a simple RFQ interaction, in which a rider shares their location and desired destination, which the service can use to estimate the cost of a journey.

In the *RFQ* protocol in Listing 1, there are two roles an agent may play: Rider and Service. The Rider can send *Request*, which has a payload of three parameters: ID, location, and destination. These parameters are adorned ⌜out⌝, which means that Rider produces new bindings for them upon sending *Request*. Service can also send one message, *Offer*, with a payload of four parameters: ID, location, destination, and price. The first three parameters are adorned ⌜in⌝, which means Service must observe a binding for them before sending *Offer*. In *RFQ*, all of the messages have ID as the only key, which uniquely identifies the enactments of *RFQ*. Each parameter may only have one binding for a given value of ID.

2.1 Message Reordering

Message reordering is a fault in several protocol specification languages, because they depend on ordering guaranties from the communication infrastructure [2].

For example, the *RFQ* protocol specified as a trace expression [3] is given as the following:

$$\textsc{Rider} \xrightarrow{\textit{Request}} \textsc{Service} \cdot \textsc{Service} \xrightarrow{\textit{Offer}} \textsc{Rider}$$

This specifies the simple requirement that a trace (log of messages) of the system match the concatenation of the traces of a single *Request* concatenated with the trace of a single *Offer*.

Although this matches exactly a single enactment of *RFQ*, since SERVICE cannot send *Offer* until it has received *Request*, there is minimal support for repeated enactments and no support for message reordering. Under Ferrando et al.'s [3] assumption of FIFO channels, if RIDER sends multiple *Request*s, it can expect to receive several *Offer* messages in the same order; multiple enactments are indistinguishable from isolated enactments. However, if something breaks and the messages are delivered out of order, RIDER would erroneously correlate the wrong *Offer* with its *Request*. Although message IDs could be automatically added by the infrastructure to help correlate requests with responses, such automatic IDs are only effective for two-party cases. A more detailed discussion of the limitations of such correlation IDs and FIFO queues is available at [2].

In contrast, systems specified by information protocols correctly associate messages with enactments without constraining the ordering, because the messages contain explicit keys that ensure correct correlation. Alternatively, information protocols are concerned with the cumulative information observed, and all information is explicitly communicated, so messages contain the same information regardless of the order they are received.

2.2 Message Duplication

Another common fault in communications is message duplication, in which a message is received multiple times, though it was sent only once.

In Section 2.1, the Trace specification of the *RFQ* protocol specifies exactly one transmission of *Request*, followed by exactly one transmission of *Offer*. If multiple *Request* are received, SERVICE will interpret them as multiple isolated enactments of *RFQ*, and presumably respond to each. Conversely, if multiple *Offer* messages are received, RIDER will incorrectly correlate them to its subsequent REQUEST messages.

In this case, the automatic addition of message IDs can identify and eliminate duplicate messages. However, relying on automatic correlation IDs imposes constraints on the infrastructure, and implicitly couples the agents to it.

Because all information in a message is explicit, information protocols are not affected by duplicate messages; they contain no new information regarding the enactment, and so do not affect the state of the agent. However, duplicate messages do potentially communicate implicit information about the environment; that something is wrong which would cause duplicates to occur. Thus, handling duplicate messages in the agent's policy instead of the infrastructure enables additional fault tolerance strategies and is important for ensuring progress despite faults, as we discuss further in Section 4.

2.3 Message Corruption

Message corruption damages the information content of a message, so information protocols alone do not provide a solution.

However, environmental message corruption can be easily detected and avoided using content-level techniques such as signatures and checksums. An unrecoverably corrupted message can be discarded and considered equivalent to message loss. As such, we do not consider accidental message corruption in detail.

2.4 Message Loss

A lost message contains no information, and is furthermore indistinguishable from delay or an agent's autonomous decision to not send the message. Therefore, an information protocol specification alone—or any protocol specification, for that matter—cannot correct message loss.

However, information protocols do enable the use of various strategies at the agent policy level for detecting and resolving message loss. The following sections discuss several causes for message loss, and strategies that can be taken to address them.

3 Internal Faults

In this section, we address faults that directly affect an agent itself, such as a bug in the agent's software implementation or a hardware failure. For simplicity, we consider that all of these faults can be abstracted to the worst case scenario of a *crash*, causing the agent to halt and cease further action. Strategies for handling lower-level faults and avoiding crashes are outside the scope of this discussion.

Figure 1 illustrates the architecture of a basic protocol-aware agent, to help identify the consequences of a crash at various points in its operation.

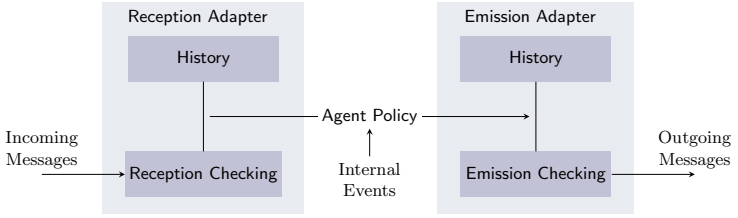

Fig. 1. An agent's internal architecture, showing how internal events and policy interact with the world through the protocol adapters.

Figure 1 shows the important components of a protocol-aware agent, and how it interacts with the world. Such an agent interacts with the world through

its protocol adapters, which keep a history of all messages received and emitted, and check outgoing messages for consistency with that history. If a message is not consistent with the agent's history it will be dropped, ensuring compliance despite faults.

The agent's internal policy processes events and emits messages through the emission adapter. Events can be external, resulting from message receptions, or internal events such as sensory perception or timers.

The points at which a crash may occur include:

1. Before logging reception
2. After logging reception, before policy
3. During policy, before logging emission
4. After logging emission, before sending

Case 1 is indistinguishable from a network connection failure. Strategies for dealing with environmental network loss are discussed in Section 4.

Cases 2 and 3 are only distinguishable if the agent is stateful; that is, if processing the message produces side effects other than message emissions. As such, we instead discuss strategies for handling crashes during policy in stateless versus stateful agent implementations.

3.1 Crash During Stateless Policy

If the agent crashes after logging the reception but before policy, a new instance can be started using the logged history information. Because the history contains all messages that the agent has received or sent, it contains all of the information necessary for a new agent to resume where it left off.

There are several ways the restarted agent can resume computation. First, the agent could simply reprocess all of the messages in its history. This approach is inefficient, but very simple. Because the agent is stateless it will produce the same output for all of its inputs. As discussed in Section 2.2, information protocols can handle duplicate messages, though they could be checked against the history to avoid sending duplicates if desired.

Another option is querying the history for enabled messages. For example, consider the Service role of the *RFQ* protocol in Listing 1. As Service is stateless, it can compute a price using only the location and destination provided by Rider. To identify prices that it needs to compute, it can search its database using the equivalent of the following SQL statement:

```
SELECT * FROM history WHERE
  location IS NOT NULL
  AND destination IS NOT NULL
  AND price IS NULL;
```

This query finds all enactments where a location and destination have been observed, but the price has not yet been computed. Service can then compute and send these missing prices.

3.2 Crash During Stateful Policy

Consider the following rideshare protocol in Listing 2, in which RIDER hires the offered ride, and SERVICE replies with a description of the dispatched vehicle.

Listing 2. The *Hire* Protocol

```
Hire {
  role Rider, Service
  parameter in ID key, in price, out payment,

  Rider -> Service: Hire[in ID key, in price, out payment]
  Service -> Rider: Ride[in ID key, in payment, out rideID key, out
      description]
}
```

In Listing 2, SERVICE dispatches a vehicle and then announces the rideID and description to RIDER. If the crash happens after dispatching the vehicle but before the ride notification, SERVICE would not be able to remember that it had dispatched the ride from its message history alone. During restart, it may dispatch a second vehicle, wasting the first driver.

In general, and especially in cases involving side effects in the real world, there will be crashes that are unrecoverable. The only solutions are low-level atomic transactions, or detailed closed-loop sensor feedback to check the state of the system before proceeding.

For situations that are recoverable in software, we propose the following normalization of the agent architecture:

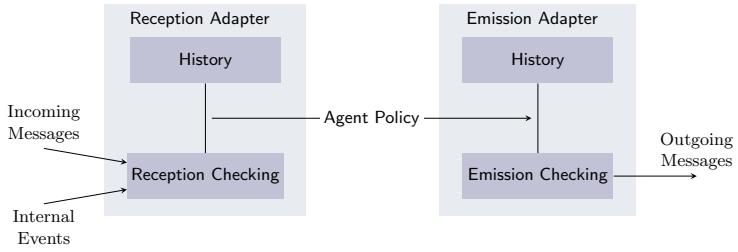

Fig. 2. Normalized agent architecture, with formerly internal events being handled as messages.

In Figure 2, the previously internal events are now treated like incoming messages, and handled by the agent's protocol adapter.

Normalizing an agent so that all of its events are handled as messages encourages proper protocol design. For example, SERVICE should be interacting with drivers via protocols. If so, those messages would be in the agent's history, and properly handled during restart.

4 External Faults

External faults are those due to environmental conditions, and therefore not the responsibility of any single agent.

Although the environment can cause any kind of error, information protocols are robust against reordering and duplication as we discussed in sections 2.1 and 2.2. Furthermore, messages corrupted by the environment are easily detected through the use of checksums and discarded, and so reduce to message loss. As such, we consider only strategies for handling message loss.

4.1 Retry Policies

If *Request* is lost during *RFQ*, it is as if the protocol had never begun—Service is not aware of the request, and so is unable to respond. However, since an enactment is identified by unique key parameters (in this case the ID field), Rider can resend the message until it gets through without confusing Service.

Conversely, if *Offer* is lost, Rider has no way to tell that Service has received its message, and so this case is indistinguishable from the first. Thus Service can expect to receive another copy of *Request* if the message was not received, and simply resend it.

This approach—the retry policy—is the basic pattern for handling message loss in information protocols, since agents may resend information without constraint. It is up to the agent's decision making when to resend a message.

This approach can also scale to larger protocols. Consider the three-party interaction in Listing 3.

Listing 3. The *Rideshare* Protocol

```
Rideshare {
  roles Rider, Service, Driver
  parameter out ID key, out loc, out dest, out payment, out rideID key, out
      description

  Rider -> Service: Hire[out ID key, out loc, out dest, out payment]
  Service -> Driver: Dispatch[in ID key, in loc, in dest, out rideID key]
  Driver -> Rider: Arrival[in ID key, in rideID key, out description]
}
```

In this simple protocol, Rider hires a ride from location to destination, Service dispatches Driver, who arrives to pick up the passenger. If all messages are transmitted successfully, no acknowledgements are required. However, even if some messages are lost, Rider can resend its request until Driver arrives.

However, retry policies depend on the closed-loop nature of the protocol. If Rider did not expect Driver to pick them up, and instead requested the ride on behalf of someone else, then they would have no way of detecting the fault and resending the message. If there is no way to detect a failure, there is no way to recover from it.

4.2 Role Replacement

An agent or connection (which are possibly indistinguishable to the other participants) may be permanently damaged, so that the only solution is to find a replacement.

At this point we discover the need for an extension to our protocol language: role adornments. Previously, roles were implicitly bound by the enactment of the protocol; perhaps all participants agreed before enacting it. However, now the selection and potential replacement of a role must be explicitly communicated within the protocol.

Listing 4. Multiple Dispatch Protocol

```
Multiple Dispatch {
  roles Rider, Service, Driver
  parameter out ID key, out loc, out dest, out payment, out rideID key, out
      description

  Rider -> out Service: Hire[out ID key, out loc, out dest, out payment]
  Service -> out Driver: Dispatch[in ID key, in Rider, out rideID key]
  Driver -> in Rider: Arrival[in ID key, in rideID key, out description]
}
```

In Listing 4, the roles are adorned ⌜in⌝ or ⌜out⌝ and may be included as parameters in a message. Specifically, Rider selects Service, who selects Driver, and Driver announces its arrival to the explicitly specified Rider.

Treating the roles as parameters explicitly specifies which roles bind the other roles, giving Service the opportunity to select a different shipper if the first proves unreliable.

5 Conclusion

In this paper, we have examined various kinds of faults that are relevant to MAS, and shown that depending on the fault information protocols either directly provide or enable strategies for both *compliance-despite-faults* and *progress-despite-faults*.

An in-depth classification of faults affecting multiagent systems has been done by Potiron et al. [7]. Our work focuses only on those faults relevant to a MAS specification, and suggests strategies for dealing with them using information protocols.

Limón et al. and Ricci et al. have discussed fault tolerance and JaCaMo [6, 8], but their suggestions have been limited to reconnecting nodes. Guessom et al. discuss fault tolerance for massive MAS [4], but focus on the architecture and replication of agents. Kumar et al. discuss fault tolerance for MAS [5], but focus on architectures for handling broker failures and recovery. None of these approaches involve specification of the MAS, let alone protocol or information protocol based specifications.

For future work, we will consider cases where the roles act maliciously, either to defraud or attack the other participants in a protocol.

Bibliography

[1] Baldoni, M., Baroglio, C., Martelli, A., Patti, V.: A priori conformance verification for guaranteeing interoperability in open environments. In: Proceedings of the 4th International Conference on Service-Oriented Computing. pp. 339–351 (December 2006)

[2] Chopra, A.K., V, S.H.C., Singh, M.P.: An evaluation of communication protocol languages for engineering multiagent systems (Oct 2019), arXiv:1901.08441v2 [cs.SE]

[3] Ferrando, A., Winikoff, M., Cranefield, S., Dignum, F., Mascardi, V.: On enactability of agent interaction protocols: Towards a unified approach. In: Proceedings of the 7th International Workshop on Engineering Multiagent Systems. p. to appear (2019)

[4] Guessoum, Z., Briot, J., Faci, N.: Towards fault-tolerant massively multiagent systems. In: Ishida, T., Gasser, L., Nakashima, H. (eds.) Massively Multi-Agent Systems I, First International Workshop, MMAS 2004, Kyoto, Japan, December 10-11, 2004, Revised Selected and Invited Papers. Lecture Notes in Computer Science, vol. 3446, pp. 55–69. Springer (2004). https://doi.org/10.1007/11512073_5, https://doi.org/10.1007/11512073_5

[5] Kumar, S., Cohen, P.R.: Towards a fault-tolerant multi-agent system architecture. In: Sierra, C., Gini, M.L., Rosenschein, J.S. (eds.) Proceedings of the Fourth International Conference on Autonomous Agents, AGENTS 2000, Barcelona, Catalonia, Spain, June 3-7, 2000. pp. 459–466. ACM (2000). https://doi.org/10.1145/336595.337570, https://doi.org/10.1145/336595.337570

[6] Limón, X., Guerra-Hernández, A., Ricci, A.: Distributed transparency in endogenous environments: The jacamo case. In: Fallah-Seghrouchni, A.E., Ricci, A., Son, T.C. (eds.) Engineering Multi-Agent Systems - 5th International Workshop, EMAS 2017, Sao Paulo, Brazil, May 8-9, 2017, Revised Selected Papers. Lecture Notes in Computer Science, vol. 10738, pp. 109–124. Springer (2017). https://doi.org/10.1007/978-3-319-91899-0_7, https://doi.org/10.1007/978-3-319-91899-0_7

[7] Potiron, K., Taillibert, P., Fallah-Seghrouchni, A.E.: A step towards fault tolerance for multi-agent systems. In: Dastani, M., Fallah-Seghrouchni, A.E., Leite, J., Torroni, P. (eds.) Languages, Methodologies and Development Tools for Multi-Agent Systems, First International Workshop, LADS 2007, Durham, UK, September 4-6, 2007. Revised Selected Papers. Lecture Notes in Computer Science, vol. 5118, pp. 156–172. Springer (2007). https://doi.org/10.1007/978-3-540-85058-8_10, https://doi.org/10.1007/978-3-540-85058-8_10

[8] Ricci, A., Ciortea, A., Mayer, S., Boissier, O., Bordini, R.H., Hübner, J.F.: Engineering scalable distributed environments and organizations for MAS. In: Elkind, E., Veloso, M., Agmon, N., Taylor, M.E. (eds.) Proceedings

of the 18th International Conference on Autonomous Agents and MultiAgent Systems, AAMAS '19, Montreal, QC, Canada, May 13-17, 2019. pp. 790–798. International Foundation for Autonomous Agents and Multiagent Systems (2019), http://dl.acm.org/citation.cfm?id=3331770

[9] Singh, M.P.: Information-driven interaction-oriented programming: BSPL, the Blindingly Simple Protocol Language. In: Proceedings of the 10th International Conference on Autonomous Agents and MultiAgent Systems. pp. 491–498 (2011)

[10] Winikoff, M., Yadav, N., Padgham, L.: A new hierarchical agent protocol notation. Autonomous Agents and Multi-Agent Systems **32**(1), 59–133 (Jul 2017)