Mandrake: Multiagent Systems as a Basis for Programming Fault-Tolerant Decentralized Applications

Samuel H. Christie V
North Carolina State University
Raleigh, NC, USA
schrist@ncsu.edu

Munindar P. Singh
North Carolina State University
Raleigh, NC, USA
mpsingh@ncsu.edu

Amit K. Chopra
Lancaster University
Lancaster, UK
amit.chopra@lancaster.ac.uk

ABSTRACT

We define a *decentralized* software application as one that consists of *autonomous* agents that communicate through *asynchronous* messaging. Constructing a decentralized application involves designing agents as independent local computations that coordinate to realize the application's requirements. Moreover, a decentralized application is susceptible to faults manifested as message loss, delay, and reordering.

We contribute *Mandrake*, a programming model for decentralized applications that addresses these challenges. Specifically, we adopt the construct of an *information protocol* that specifies messaging between agents purely in causal terms and can be correctly enacted by agents in a shared-nothing environment over nothing more than unreliable, unordered transport. Mandrake facilitates (1) implementing protocol-compliant agents by introducing a programming model; (2) transforming fragile protocols into fault-tolerant ones with simple annotations; and (3) a declarative policy language that makes it easy to implement fault-tolerance in agents based on the capabilities in protocols. In obviating the reliance on reliability and ordering guarantees in the communication infrastructure, Mandrake achieves some of the goals of the founders of networked computing from the 1970s.

CCS CONCEPTS

• **Computing methodologies → Multi-agent systems**; • **Software and its engineering → Software fault tolerance**.

KEYWORDS

Communication protocols, Asynchrony, Programming model

ACM Reference Format:
Samuel H. Christie V, Munindar P. Singh, and Amit K. Chopra. 2023. Mandrake: Multiagent Systems as a Basis for Programming Fault-Tolerant Decentralized Applications. In *Proc. of the 22nd International Conference on Autonomous Agents and Multiagent Systems (AAMAS 2023), London, United Kingdom, May 29 – June 2, 2023*, IFAAMAS, 3 pages.

1 INTRODUCTION

We observe two motivations for decentralization. First, decentralization reflects autonomy in the overarching social architecture of an application [3]. Applications in domains such as finance and healthcare span multiple autonomous real-world parties. Second, decentralization reflects loose coupling in the technical architecture. *Microservices* [7] support developing, deploying, and scaling microservices independently of each other. The Internet of Things (IoT) motivates decentralization in both social and technical terms [6, 11] by bringing forth interactions between devices owned by two or more parties and by technologies such as fog computing that distribute information processing and storage [8].

However, getting decentralized applications right is extremely difficult. Asynchrony and faults make coordinating the computations of an application challenging. This challenge is exacerbated when, as in open applications, agents represent autonomous real-world parties and are independently constructed. Further, autonomy motivates flexibility in interactions [13]; however, flexibility itself is in tension with ease of coordination [2, 12].

Conventional approaches for building distributed systems assume reliable and ordered delivery communication services, e.g., as exemplified by TCP and message queues [1]. As the end-to-end model [9] anticipated however, baking coordination mechanisms into the infrastructure interferes with application-level decision making and hits performance. Traditional fault tolerance is not particularly meaningful because it operates transparently to the application, without influencing or considering agent decisions.

This paper addresses the challenges of building robust decentralized applications in a manner compatible with the end-to-end model. Our contribution, *Mandrake*, is a set of techniques that show how a fault-tolerant decentralized application can be realized as a multiagent system sitting on top of an infrastructure that provides neither ordering nor reliability guarantees. Mandrake is based on the insight that to support meaning, a decentralized application must be modeled via a declarative information protocol [10]. An information protocol captures the interactive part of application meaning by constraining an agent's emission of messages based purely on its local information state, which is the agent's history of communications. Receptions, however, are not constrained. Specifically, an agent may receive messages sent by others in any order. An agent's decision making (which would generally rely both on its local state and internal state) is up to the agent's implementation. Taken together, the possibility of an agent receiving messages in any order and processing them in accordance with its own decision making are crucial to realizing application meaning.

Such an application architecture provides an opportunity to introduce meaningful fault tolerance—that is, fault tolerance based on information available to agents and incorporated in their decision making. In particular, it enables understanding a fault as the violation of an *expectation* of an agent to receive some information from others and fault tolerance as what agents do to prevent or handle the violations.

Proc. of the 22nd International Conference on Autonomous Agents and Multiagent Systems (AAMAS 2023), A. Ricci, W. Yeoh, N. Agmon, B. An (eds.), May 29 – June 2, 2023, London, United Kingdom. © 2023 International Foundation for Autonomous Agents and Multiagent Systems (www.ifaamas.org). All rights reserved.

2 SCENARIO

To illustrate our ideas, Listing 1 specifies a prescription scenario as an information protocol called Treatment.

Listing 1: Treatment Protocol.

```
Treatment {
  roles Patient, Doctor, Pharmacist
  parameters out sID key, out symptom, out done

  Patient -> Doctor: Complaint[out sID key, out
      symptom]
  Doctor -> Patient: Reassurance[in sID key, in
      symptom, nil Rx, out done]
  Doctor -> Pharmacist: Prescription[in sID key,
      in symptom, nil done, out Rx]
  Pharmacist -> Patient: FilledRx[in sID key, in
      Rx, out done]
}
```

In the scenario, Patient can send a complaint to Doctor, who sends a prescription to Pharmacist (or reassures Patient that they don't need medication), who fills it and sends a notification back to Patient, completing the interaction. Importantly, each agent may only send a message if the specified causality and integrity constraints are satisfied.

In this scenario, any of the messages could fail to arrive. This is the fundamental interaction fault; all failures in sending or transmitting a message are reducible to message loss. If any of the messages in the Treatment protocol do not arrive, the interaction cannot be completed since agents will lack information necessary for further actions.

3 APPROACH

To make progress despite such faults, agents must be able to detect and compensate for them. We propose an application-level focus on fault tolerance. Our approach consists of two main aspects:

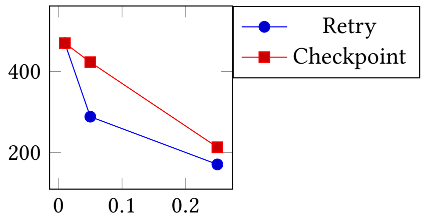
First, a protocol specification is used to identify the *expectations* that each agent can have based on the information it should observe in a correctly progressing interaction. These expectations are the basis of detecting faults; if an expectation is not met within a reasonable (application and possibly even enactment specific) time-frame, a fault may be assumed to have occurred.

Secondly, we propose protocol transformations and a policy language to support the implementation of detection and recovery policies. Although fault tolerance should be handled at the application level, that does not mean that every application developer should waste time redesigning basic recovery policies. Instead, off-the-shelf patterns and simple tools can help simplify solving the problem without overly burdening application developers.

The protocol transformations add messages to a protocol that propagate information between agents. For example, a forwarding transformation could be used to let Doctor send Patient a copy of the prescription, and then allow Patient to forward it again to Pharmacist directly.

If such forwarding transformations are applied to the Treatment protocol, then a robust recovery policy such as the one in Listing 2 could be used.

Figure 1: Unreliable Doctor



Listing 2: A robust recovery policy

```
// Doctor
- action: forward Prescription
// Patient
- action: remind Doctor of Complaint until
    Reassurance or Prescription
  when: 0 12 * * * // daily at noon
- action: remind Pharmacist of Prescription after
    2 days until FilledRx
  when: 0 0 * * * // daily
  max tries: 5
// Pharmacist
- action: remind Patient of FilledRx upon
    Prescription reminder
```

This policy enables Patient to directly act to achieve its goals when its expectations for receiving FilledRx are unmet. Furthermore, Patient can take the most direct path toward its goal: if it has not yet received a copy of the prescription, then it must remind Doctor, but if it has then it can more efficiently and directly send Prescription to Pharmacist. This can be especially helpful if only Doctor is unreliable.

4 CONCLUSION

We ran simulations to verify the difference that various policies made in the reliability and performance of this system, under different types of load and message loss. The most dramatic result is shown in Figure 1, which compares the rate at which enactments are completed per second at different message loss probabilities between two recovery policies. The blue/circle line shows a simple retry policy, where Patient merely reminds Doctor about its complaint to trigger the rest of the interaction again, and the red/square line shows the checkpoint policy given above where Patient can forward information directly to Pharmacist if available.

The full paper [4] describes our approach and our results in detail. Since then, we have extended the programming model to support decision making [5]. Our work demonstrates that application-specific fault tolerance policies, using interaction protocol specifications as a foundation, have the potential to improve performance and reliability without burdening developers in common cases.

ACKNOWLEDGMENTS

We thank the anonymous reviewers for their comments. We thank the EPSRC (grant EP/N027965/1, Turtles) and the US National Science Foundation (grant IIS-1908374) for partial support.

REFERENCES

[1] AMQP. 2014. Advanced Message Queuing Protocol. http://www.amqp.org Accessed 2022-03-28.

[2] Matteo Baldoni, Cristina Baroglio, Amit K. Chopra, Nirmit Desai, Viviana Patti, and Munindar P. Singh. 2009. Choice, Interoperability, and Conformance in Interaction Protocols and Service Choreographies. In *Proceedings of the 8th International Conference on Autonomous Agents and MultiAgent Systems (AAMAS)*. IFAAMAS, Budapest, 843–850. https://doi.org/10.5555/1558109.1558129

[3] Amit K. Chopra and Munindar P. Singh. 2016. From Social Machines to Social Protocols: Software Engineering Foundations for Sociotechnical Systems. In *Proceedings of the 25th International World Wide Web Conference*. ACM, Montréal, 903–914. https://doi.org/10.1145/2872427.2883018

[4] Samuel H. Christie V, Amit K. Chopra, and Munindar P. Singh. 2023. Mandrake: Multiagent Systems as a Basis for Programming Fault-Tolerant Decentralized Applications – Journal-First Abstract. In *Proceedings of the 22nd International Conference on Autonomous Agents and MultiAgent Systems (AAMAS)*. IFAAMAS, London, 1–2.

[5] Samuel H. Christie V, Munindar P. Singh, and Amit K. Chopra. 2023. Kiko: Programming Agents to Enact Interaction Protocols. In *Proceedings of the 22nd International Conference on Autonomous Agents and MultiAgent Systems (AAMAS)*. IFAAMAS, London, 1–9.

[6] Henning Kagermann, Wolfgang Wahlster, , and Johannes Helbig. 2013. Recommendations for implementing the strategic initiative INDUSTRIE 4.0: Securing the future of German manufacturing industry.

[7] Irakli Nadareishvili, Ronnie Mitra, Matt McLarty, and Mike Amundsen. 2016. *Microservice architecture: aligning principles, practices, and culture*. O'Reilly Media, Inc.

[8] Frank Pallas, Philip Raschke, and David Bermbach. 2020. Fog Computing as Privacy Enabler. *IEEE Internet Computing* 24, 04 (jul 2020), 15–21.

[9] Jerome H. Saltzer, David P. Reed, and David D. Clark. 1984. End-To-End Arguments in System Design. *ACM Transactions on Computer Systems* 2, 4 (Nov. 1984), 277–288. https://doi.org/10.1145/357401.357402

[10] Munindar P. Singh. 2011. Information-Driven Interaction-Oriented Programming: BSPL, the Blindingly Simple Protocol Language. In *Proceedings of the 10th International Conference on Autonomous Agents and MultiAgent Systems (AAMAS)*. IFAAMAS, Taipei, 491–498. https://doi.org/10.5555/2031678.2031687

[11] Munindar P. Singh and Amit K. Chopra. 2017. The Internet of Things and Multiagent Systems: Decentralized Intelligence in Distributed Computing. In *Proceedings of the 37th IEEE International Conference on Distributed Computing Systems (ICDCS)*. IEEE, Atlanta, 1738–1747. https://doi.org/10.1109/ICDCS.2017.304 Blue Sky Thinking Track.

[12] Michael Winikoff. 2007. Implementing Commitment-Based Interactions. In *Proceedings of the 6th International Joint Conference on Autonomous Agents and MultiAgent Systems (AAMAS)*. IFAAMAS, Honolulu, 868–875. https://doi.org/10.1145/1329125.1329283

[13] Pınar Yolum and Munindar P. Singh. 2002. Commitment Machines. In *Proceedings of the 8th International Workshop on Agent Theories, Architectures, and Languages (ATAL 2001) (LNAI, Vol. 2333)*. Springer, Seattle, 235–247.