Bungie: Improving Fault Tolerance via Extensible Application-Level Protocols

Samuel H. Christie V
Lancaster University and North Carolina State University

Amit K. Chopra
Lancaster University

Munindar P. Singh
North Carolina State University

***Abstract*—Decentralized applications involve interactions between autonomous agents. As such, they must contend with *communication* faults, such as message loss. However, due to the inadequacy of existing application programming models, current approaches leave fault handling to the infrastructure, e.g., by employing reliable transport protocols.**
In contrast, we demonstrate how causality and information-based design principles can reveal an application's exposure to communication faults, and thereby enable the deployment of application-specific strategies for recovering from faults. We present *Bungie*, an approach based on application-level protocols that precisely capture the causality inherent to the interactions between agents. We show through patterns and examples how Bungie provides abstractions for achieving fault tolerance.

Index Terms: Protocol; Autonomy; End-to-End Argument; Application Meaning; Fault Tolerance; Agents

1. Introduction

We are concerned with *decentralized* applications, whose computations are performed by autonomous and heterogeneous *agents*. The agents represent real-world autonomous parties such as people or organizations, and could be manifested in hardware (e.g., delivery drones) or software (e.g., microservices). Further, the agents coordinate their computations via asynchronous messaging, without relying on any central store of state. Multiparty engagements in e-business [1], health [2], and finance [3] already emphasize messaging and naturally lend themselves to decentralization. We demonstrate our ideas with a pharmacy scenario, in which a patient presents symptoms to a doctor who prescribes medication that is filled by a pharmacist.

Communication provides a suitable basis for modeling the interactions of autonomous agents. Since the dawn of networking, methods for developing distributed applications have contended with communication faults such as message loss. Traditional distributed computing favors handling faults in the communication infrastructure, invisibly to the application.

An alternative approach is to base fault handling on *application meaning*, which respects the

famous end-to-end principle [4]. In our pharmacy scenario, what matters to the application's reliability is that the pharmacist fills a prescription, not merely that a prescription is delivered to the pharmacist by the infrastructure. Therefore, any fault handling in the infrastructure is unnecessary and only imposes costs—e.g., delivering prescriptions in FIFO order (as TCP does) even though the pharmacist would fill them based on availability and urgency. Clark [5] reiterates the importance of application meaning.

Further, to reduce coupling between agents and improve reuse and portability, we avoid common assumptions of the infrastructure such as its reliability or the channels being ordered.

We investigate this question: **How can we leverage application meaning to achieve fault tolerance while ensuring flexibility and reusability of decentralized applications?** We demonstrate an approach, dubbed Bungie, based on application-level *information protocols* (see sidebar) [6]. A protocol captures only the information exchanged by an application's agents and does not capture or interfere with their internals. Specifically, we make the following contributions.

- Defining agent expectations (the messages an agent expects to receive) based on information protocols and explaining how unmet expectations capture faults.
- Describing agent policies for recovering from faults and how policies fit into a protocol-based agent architecture.
- Demonstrating protocol extension patterns that can augment a given protocol with fault tolerance capabilities such as forwarding and acknowledgments.

Sidebar: Information Protocols

An information protocol specifies a multiagent application abstractly in terms of the roles in the application, messages exchanged by the roles, and constraints on those messages. In a concrete application, each role is bound to an agent. To illustrate the relevant ideas, we specify the prescription scenario as an information protocol using the Blindingly Simple Protocol language (BSPL) in Listing 1.

In Listing 1, the roles are DOCTOR, PATIENT, and PHARMACIST; the messages are *Complain*, *Prescribe*, and *Filled*.

Each message specifies the information it conveys via parameters. For example, *Complain*'s parameters are vID and symptoms, which represent the unique visit ID and a description of the patient's symptoms, respectively.

Listing 1: Prescription protocol in BSPL

```
Prescription {
  roles Doctor, Pharmacist, Patient
  parameters out vID key, out Rx, out package

  Patient -> Doctor: Complain[out vID key, out
      symptoms]
  Doctor -> Pharmacist: Prescribe[in vID key, in
      symptoms, out Rx]
  Pharmacist -> Patient: Filled[in vID key, in
      Rx, out package]
}
```

Agents enact a protocol by sending and receiving message instances. Protocol enactments are constrained by *integrity* and *causality* constraints.

Integrity constraints on enactments are specified by annotating one or more parameters of each message as *key* parameters. A tuple of bindings for the key parameters identifies a message instance. Consistent correlation across message instances requires that, for each tuple of bindings for key parameters, there can be at most one binding for a nonkey parameter. For example, in Listing 1, all messages have the same key parameter, vID. Therefore, vID identifies instances of each message. Further, if an instance of *Complain* has vID and symptoms bound to `1` and `cough`, respectively, then any instance of *Prescribe* with vID bound to `1` must also have symptoms bound to `cough`.

Causality constraints are information dependencies and are indicated by adorning every parameter as ⌜out⌝, ⌜in⌝, and ⌜nil⌝. The adornments ⌜out⌝ and ⌜nil⌝ mean that the parameter's binding must not be known in the enactment, though with ⌜out⌝ the binding is generated; ⌜in⌝ means that the binding must be known. For example, the adornments ⌜out⌝ for vID and symptoms in *Complain* mean that PATIENT can send an instance of *Complain* by generating bindings for both parameters. The adornments ⌜in⌝ for vID and symptoms in *Prescribe* mean that to send an instance of *Prescribe*, DOCTOR must already know their bindings (in our scenario, from receiving a *Complain*); however, DOCTOR can generate any binding it wants for Rx since that parameter is adorned ⌜out⌝.

2. Modeling Faults as Unmet Expectations

An application must be able to detect a communication fault to recover from it. An agent can notice the absence of a message only if it expected to receive that message.

Faults modeled as unmet expectations capture application requirements. At run-time, expectations help detect possible message loss even without direct knowledge of the transmission attempt. For example, the patient expects to receive *Filled* after sending *Complain*, even without knowing that *Filled*'s transmission has been attempted. During design, expectations also identify the responsibilities each agent has for recovering from a loss. For example, the patient (who alone has expectations for the prescription) must initiate any recovery, but the others should cooperate.

Expectations are essential to many protocol languages (including BPMN [7], Scribble [8], and trace expressions [9]) because they specify protocols as sequences of events: each agent expects the next event(s) in the sequence. However, the obviousness of expectations in these languages also limits their flexibility; because agents are told exactly what to expect, there is no room for unexpected events such as a fault or corresponding recovery. Modeling pervasive, flexible alternatives in such languages is nontrivial [10].

Conversely, information protocols are naturally flexible because they constrain events by their causal information dependencies and leave the emission up to the agent. This flexibility comes at the cost of nontrivial expectations, but we can still compute them.

Given a protocol, we say that an agent expects a message instance if it has observed the message's key parameters and can receive the message. The keys are necessary because expectations are for specific message instances, which can only be identified by their key parameters. An agent can receive a message if they are the specified recipient, though further deduction may refine their expectations based on observations of prerequisites or contrary messages.

Consider Listing 1. Before an enactment is initiated, no agent expects anything—they have not observed any key parameters, and DOCTOR has no basis for predicting when PATIENT may become ill. After PATIENT sends *Complain* with some vID, it can expect to observe a *Filled* message with that vID because it has observed the keys and there aren't any potential conflicts involving the other parameters. However, neither of the other agents ever expects to receive anything, because they are not aware of any enactments until they have received the only message they can.

Besides the directly expected message, an expectation *covers* all necessary intermediate messages; if any of those messages are lost, dependencies will not be satisfied, messages will not be enabled, and the expectation will be unmet. In *Prescription*, *Filled* depends on *Prescribe* so PATIENT's expectation of *Filled* covers *Prescribe*; if either message is lost, PATIENT will notice because of its expectation for *Filled*.

Any messages that are not covered by an expectation could be lost without any agent noticing. Such uncovered messages are a 'design smell' that should be tolerated only rarely. Either messages should be added to cover them and enable recovery, or the uncovered messages should be removed if they are truly unimportant.

2.1. Detecting Unmet Expectations at Runtime

Expectations are an application-level concept, requiring the designer to configure which expectations are important, when they should be triggered, and how much delay to tolerate.

Here is a sketch of an algorithm for each agent to detect unmet expectations during an enactment:

- Wait until all of the key parameters for a message reception have been observed
- Set a timer for the expectation based on application requirements (e.g., 2 business days to fill a prescription)
- Optionally update the timer when messages on which the expectation depends are observed
- When the timer expires, declare the expectation unmet.

In an enactment of *Prescription*, PATIENT expects to eventually receive *Filled* after observing vID, *Filled*'s only key. PATIENT binds vID when sending *Complain*, causing PATIENT to also set a timer for the expectation. If PATIENT does not receive *Filled* after two business days, it considers the expectation unmet.

2.2. Responding to Unmet Expectations at Runtime

In Bungie, an agent responds to unmet expectations via plugins called *external policies*. These plugins determine when (and how often) information is sent to ensure the interaction proceeds.

Bungie supports two types of external policies: *Transmission Policies* apply to any protocol, whereas *Protocol-Specific policies* work in concert with particular protocol patterns. Neither of these types of policy affects an agent's internals.

3. Information-Based Fault Tolerance

We adopt application-level information protocols (see sidebar) as the basis for modeling an interaction, and detecting and recovering from faults that may arise in an enactment.

3.1. Concepts

In information protocols the meaning of a message is determined by its parameters and correlated by its keys, giving each message an idempotent effect on the information that an agent observes. As such, messages can be received and interpreted correctly at any time, even if they are reordered or duplicated. For example, if DOCTOR received two *Complain* messages with the payload `[1,cough]` it would ignore the second as a duplicate because they have the same vID; most protocol languages would treat them as separate enactments, because *Complain* initiates enactments of *Prescription*. These features of information protocols make them suited for asynchronous, unreliable communication such as UDP.

Unfortunately, not all communication faults are as benign as reordering or duplication, and two additional capabilities are needed for tolerating communication-related faults:

- Identifying potential message loss; and
- Recovering from message loss.

In general, message loss can be addressed only through retransmission. Protocol-derived expectations can help identify loss; our protocol semantics helps identify what information is missing and how to produce or communicate it either directly or in cooperation with other agents.

3.2. Architecture

Figure 1 shows our proposed agent architecture. An agent that is able to participate in an information protocol has four essential components: incoming and outgoing interfaces with the communication infrastructure (Receiver and Emitter), internal construction (Agent Internals), and a local representation of the state of its ongoing interactions with other agents (Local State & Checker). To this we add two components geared toward storing and executing policies to promote fault tolerance.

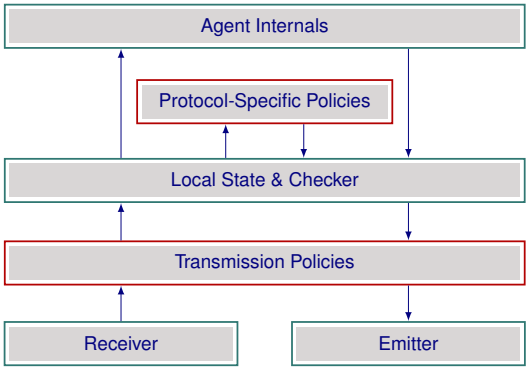


Figure 1: Bungie architecture highlighting the two policy components introduced in Bungie to improve fault tolerance.

Agent Internals are the internal databases and decision-making logic of an agent, never to be exposed to other agents lest the agents become tightly coupled.

The *Local State & Checker* records the messages and parameters that an agent has observed. The Checker is configured by the protocol specification, and checks messages to ensure they satisfy its integrity and causality constraints before they are recorded to the local state. Checking on reception is only necessary if the agents do not trust each other.

The *Receive* and *Emitter* encapsulate network transmission and processing.

The *Transmission Policies* operate between the receiver and emitter and the local state, because they handle how a message is physically packaged, conveyed, and reconstructed through the infrastructure. Discussed in Section 4.

Protocol-Specific Policies rely on the availability of specialized message schemas in the protocol, discussed in Section 5. Protocol-specific

policies actively send messages instead of managing the transmission of messages that the agent already sent, though they only use information already available in the local state.

4. Recovering from Message Loss with Transmission Policies

We propose two examples of transmission policies, Resend and Content.

4.1. Resend Policies

Assuming that disruptions causing unmet expectations are temporary, an agent can eventually recover from a fault by resending the information; this is especially the case for an application designed using information protocols, in which duplicate messages are idempotent.

Unlike infrastructure-level guarantees that an emitted message will be received, expectations also encompass application requirements that a message be produced in the first place. PATIENT expects *Filled* after sending *Complain*. It is not enough for DOCTOR to successfully receive *Complain*; DOCTOR and PHARMACIST must also produce and transmit *Prescription* and *Filled* to satisfy PATIENT's expectation.

Thus, there are two cases in which an application-level Resend policy resends messages: first, in response to an unmet expectation, and second, in cooperation with another agent's unmet expectations. In either case, the agent should resend all *relevant* messages; i.e., those that may help satisfy the expectation.

Responding to Unmet Expectations Resending a message in response to an unmet expectation clearly promotes progress if that message was the one that was lost. If *Complain* is lost when PATIENT first sends it, DOCTOR will not produce a prescription, and the prescription will not be filled, violating PATIENT's expectation for *Filled*. If PATIENT's Resend policy responds to the unmet expectation by resending *Complain*, it will eventually (assuming temporary disruption) get through, recovering from the loss.

Note that DOCTOR does not mistake a duplicate *Complain* for a new *Complain*, because each *Complain* is uniquely identified by vID.

Information protocols uniquely support the technique of sending duplicate messages to recover from loss, because they inherently provide the idempotence and emission flexibility required.

Cooperating after Receiving a Duplicate If *Prescribe* or *Filled* were lost instead, however, resending *Complain* alone would not be enough; DOCTOR and PHARMACIST must cooperate by resending their messages to satisfy *Patient*'s unmet expectation. DOCTOR can detect that some expectation is likely unmet if it receives a duplicate message. Then, DOCTOR cooperates by resending all relevant messages—those that depend on the parameters in the duplicate message, since the precise expectation may be unclear.

Agents that capture such a Resend policy would cooperate until all losses are recovered. This policy additionally enables recovery from some agent failures, because it resends *all* relevant information, making up for any gaps in the agents' local states due to a crash, as encouraged by the fail-fast [11] or crash-only [12] paradigms.

4.2. Content Policies

Content policies adjust the actual data being transmitted to improve reliability or performance, without changing the meaning of the protocol.

To facilitate content policies, we introduce the *bun*, a representation of a message instance prepared for transmission or storage. Buns are also distinct from network packets, because multiple buns may be sent in a single packet, or one bun may be split across several packets.

Content policies add or remove parameters from buns to guard against potential loss, or reduce overhead.

Redundancy adds copies of parameters from other messages to a bun to ensure that they are received.

Listing 2 shows a prescription refill protocol. In *Refill*, PATIENT requests a refill, and receives a quote for the price. PATIENT may then send the delivery address or payment in any order.

Listing 2: Prescription Refill

```
Refill {
  roles Patient, Pharmacist
  parameters out RxID key, out Rx, out price, out
      address, out payment, out package

  Patient -> Pharmacist: Request[out RxID, out Rx]
  Pharmacist -> Patient: Quote[in RxID, in Rx,
      out price]
```

```
  Patient -> Pharmacist: RequestDelivery[in RxID,
      out address]
  Patient -> Pharmacist: Pay[in RxID, in price,
      out payment]
  Pharmacist -> Patient: Delivery[in RxID, in Rx,
      in payment, in address, out package]
}
```

Under Redundancy, PATIENT's transmission policies modify the bun used to transmit the *Pay* message to include the address. Consequently, even if the *RequestDelivery* message instance is lost, PHARMACIST will still receive all of the information.

Listing 3 shows the bun resulting from PATIENT applying Redundancy to an instance of *Pay* before sending it to PHARMACIST. We write => instead of -> to differentiate a bun from a message schema.

Listing 3: An example of Redundancy

```
Patient => Pharmacist: Pay[RxID, price,
    payment, address]
```

Redaction reduces the information encoded in a bun. Redaction policies can be applied during either message emission or reception. The sender can redact on emission to remove parameters that the recipient has already received, saving bandwidth. The recipient may also redact on reception, dropping parameters they won't use and saving space in the local state.

In *Refill*, the *Quote* message has Rx as ⌜in⌝. Normally ⌜in⌝ parameters are included in the bun, but since the recipient (PATIENT) has already observed this parameter, PHARMACIST can redact it and send only the essential keys and price. Listing 4 shows an example of Redaction, where Rx has been removed from *Quote*

Listing 4: An example of Redaction

```
Pharmacist => Patient: Quote[RxID, price]
```

Alternatively, PHARMACIST can send *Quote* as usual and let PATIENT drop the duplicated *Rx* parameter on reception to reduce storage.

These performance enhancements illustrate the flexibility afforded through an information protocol. Without an information protocol, messages are opaque and must be sent verbatim since the effects of any modification are unknowable. An information protocol, however, makes the information known to agents clearer, enabling the construction of buns that may include or exclude redundant information.

A sophisticated implementation could dynamically adjust the redundancy or redaction in response to network congestion and loss rates.

5. Recovering from Loss with Protocol Extensions

Though simple and effective, transmission policies operate under the restrictions of the existing protocol. If no agent expects a given message, its loss cannot be detected. If an agent can detect the loss but has no relevant messages to send, it has no means of recovery. Even when recovery is possible, it may require cooperation from multiple agents.

Protocol extensions can help by adding messages to a protocol, enabling new expectations or means of recovery. Because protocol extensions add messages, the agents must be extended to support them through what we term *protocol extension policies*, a subset of protocol-specific policies. Since these policies do not produce new information, they can be plugged into an agent without modifying its existing behavior.

The agent developer is responsible for extending the protocol and agent. Although we can produce tools that perform the extension, the design is necessarily application-specific and must be done by a designer with knowledge of application requirements.

5.1. Forwarding

A Forwarding extension inserts a message to *forward* an existing message to a new recipient. Forwarding in this context is not limited to passing messages on from the original recipient, but encompasses any new transmission of the original message, such as from the original sender to a new recipient, or from a new recipient to the original recipient. A forwarded message can enable new expectations or provide new ways to respond to an unmet expectation.

Listing 5 shows the *RxForward* protocol, which extends *Prescription* to enable DOCTOR to forward copies of *Prescribe* messages to PATIENT, and also enables PATIENT to forward them to PHARMACIST. We write a forwarding schema with the name of the original schema prefixed with a hyphen.

Listing 5: Forwarding extension

```
RxForward {
  roles Doctor, Patient, Pharmacist
  parameters out vID key, out symptoms, out Rx,
       out package

  Patient -> Doctor: Complain[out vID key, out
       symptoms]

  Doctor -> Pharmacist: Prescribe[in vID key, in
       symptoms, out Rx]
  Doctor -> Patient: -Prescribe
  Patient -> Pharmacist: -Prescribe

  Pharmacist -> Patient: Filled[in vID key, in
       Rx, out package]
}
```

To take advantage of the extended protocol, DOCTOR should forward *Prescribe* (that is, send *-Prescribe*) to PATIENT to guard against the possibility of *Prescribe* or *Filled* being lost. After receiving *-Prescribe*, PATIENT can respond to an unmet expectation by sending *-Prescribe* to PHARMACIST, providing enough information for PHARMACIST to send (or resend) *Filled* regardless of which message was lost.

The Forwarding extension and policy reduce complexity and latency compared to Resend policies alone. Resend policies often work indirectly, requiring multiple messages to satisfy expectations, such as when PATIENT resends *Complain* to DOCTOR even though its unmet expectation is for *Filled* from PHARMACIST. However, the two policies are not incompatible; PATIENT could employ both at the same time for redundancy, or forward *-Prescribe* if it is available and otherwise resend *Complain*.

5.2. Acknowledgment

An Acknowledgment extension adds messages that acknowledge receipt of some message.

Requiring acknowledgments for every message, as is often done at the infrastructure level, can be quite wasteful if losses are rare. Further, infrastructure acknowledgments only indicate that the infrastructure has handled the message, not that the agent has acknowledged it in a meaningful sense. For example, a prescription refill request may be reliably submitted through a web portal because of infrastructure acknowledgments, but only a confirmation email or page truly indicates the pharmacist's legal commitment to fulfill the request. Bungie supports both sending meaningful acknowledgments, and using them for application-level fault tolerance.

Listing 6 shows the *Prescribe* protocol extended with acknowledgment messages.

Listing 6: Acknowledgments Extension

```
RxAck {
  roles Doctor, Pharmacist, Patient
  parameters out vID key, out symptoms, out Rx,
       out package

  Patient -> Doctor: Complain[out vID key, out
       symptoms]
  Doctor -> Patient: @Complain

  Doctor -> Pharmacist: Prescribe[in vID key, in
       symptoms, out Rx]
  Pharmacist -> Doctor: @Prescribe

  Pharmacist -> Patient: Filled[in vID key, in
       Rx, out package]
  Patient -> Pharmacist: @Filled
}
```

Acknowledgments need contain only the key parameters, which identify the instance being acknowledged.

To take advantage of the Acknowledgment extension, recipients should employ an Acknowledgment policy that acknowledges messages when they are received and recorded in the local state. The senders can respectively expect acknowledgments in response to the messages they send, and apply a Resend policy to repeatedly resend the message until that expectation is met.

Employing the Acknowledgment strategy at the protocol level does not interfere with the ability to send or receive other messages. Where TCP would block a stream until the next packet in the sequence has been received, Bungie agents can resend messages that have not been acknowledged without blocking other messages. This problem, called head-of-line blocking, was one of the primary motivations for the QUIC protocol [13] used in HTTP/3.

5.3. Probing

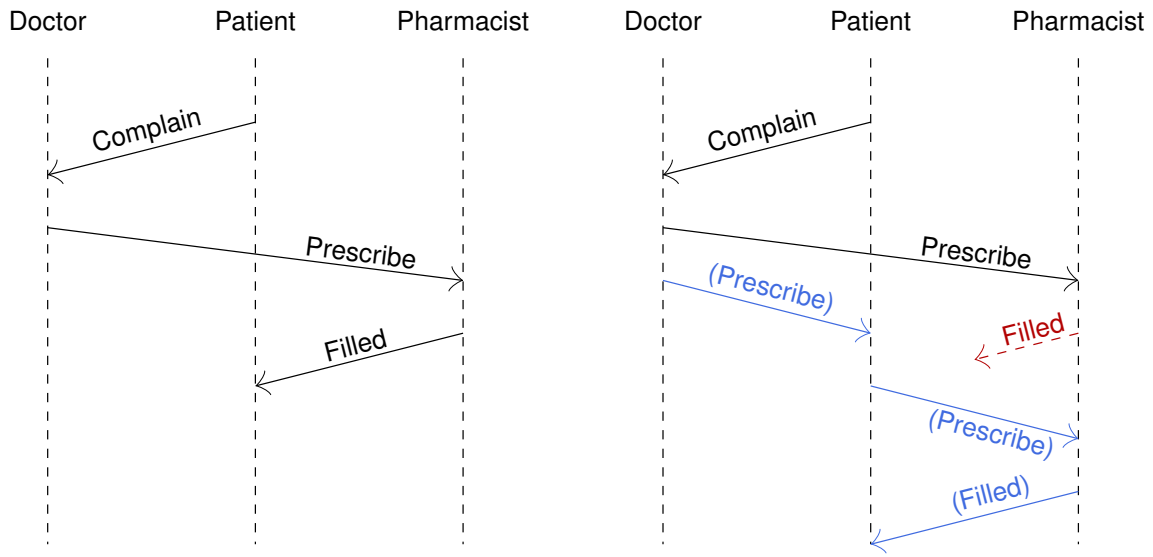
A Probing extension adds messages that enable an agent to query another for information, i.e., for messages that match a particular key.

Listing 7: Probing extension

```
RxProbe {
  roles Doctor, Patient, Pharmacist
  parameters out vID key, out symptoms, out Rx,
       out package

  Patient -> Doctor: Complain[out vID key, out
       symptoms]
  Doctor -> Pharmacist: Prescribe[in vID key, in
       symptoms, out Rx]
  Pharmacist -> Patient: Filled[in vID key, in
       Rx, out package]

  Patient -> Pharmacist: probe[in vID key]
```



(a) Original enactment.

(b) Recovering from a fault.

Figure 2: A protocol extension in use. With Forwarding extensions, DOCTOR forwards *Prescribe* to PATIENT and PATIENT forwards *Prescribe* to PHARMACIST. PHARMACIST resends *Filled*, which gets through the second time. For accessibility, we place the extended messages in parentheses and lost messages on dashed lines to highlight them without relying on color.

```
  Pharmacist -> Doctor: probe[in vID key]
  Doctor -> Patient: probe[in vID key]
}
```

In the *RxProbe* protocol in Listing 7, the Probing extension starts with PATIENT, which knows vID after sending *Complain*. According to the Probing extension pattern, the protocol is extended so that each agent has a new probe message based on the keys that it can observe.

With an appropriate Probing policy, PATIENT can send a probe to PHARMACIST when its expectation for *Filled* is unmet. PHARMACIST then either responds to the probe by resending the relevant messages that it has observed, or by sending its own probe on to DOCTOR.

6. Discussion

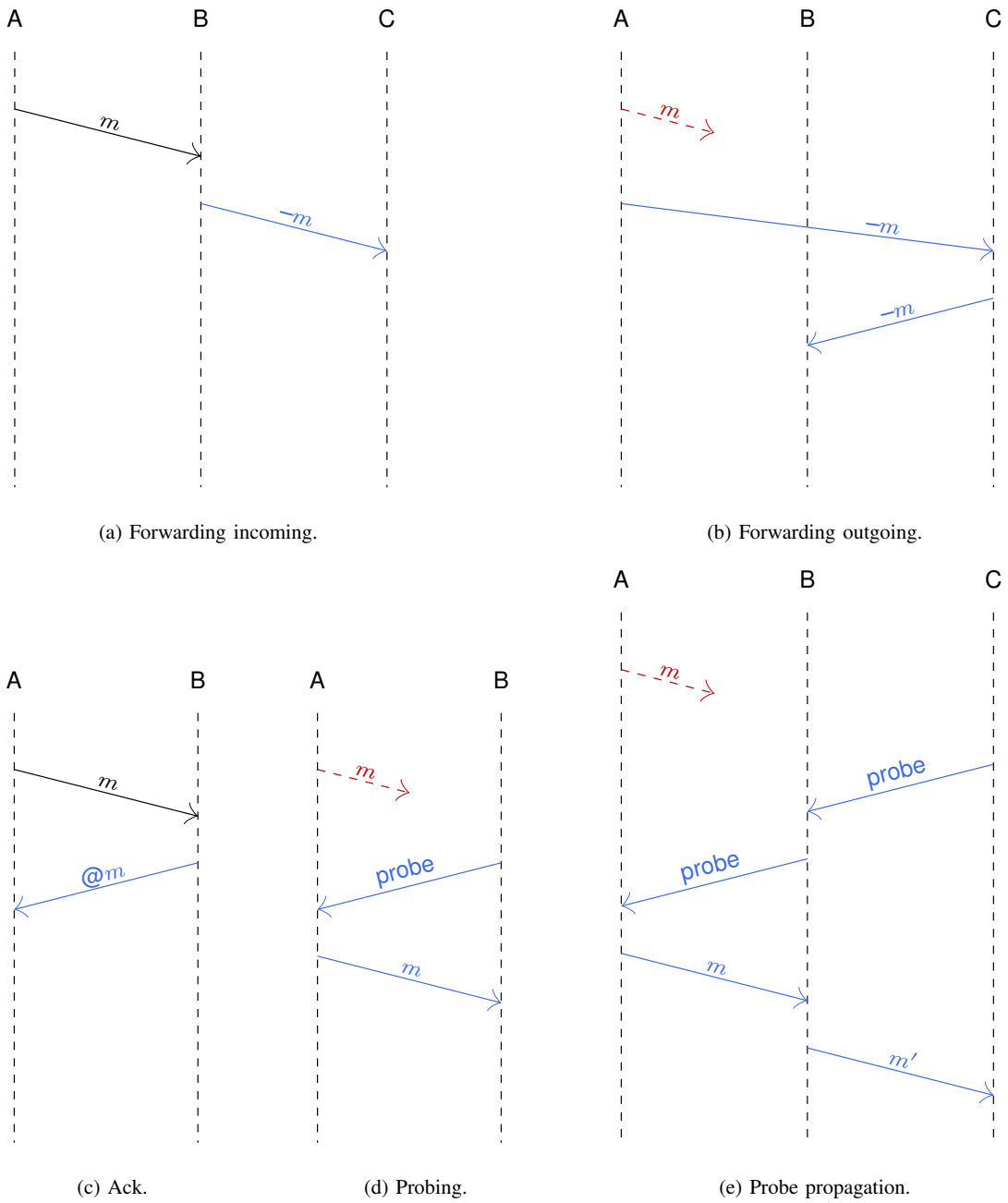
We have presented a variety of policies to illustrate how information protocols support fault tolerance at the application level, and specifically in the endpoints of the application.

Ideally, the development overhead can be reduced with a library of common configurable policies. Any situations that cannot be accommodated by common policies further demonstrate the need for an architecture like Bungie that supports application-specific fault tolerance policies.

Bungie's ability to improve fault tolerance for applications relies on two assumptions. First, that the disruptions causing message loss are temporary; and second, that the agents cooperate with the chosen recovery policies. These assumptions are reasonable; infrastructure-level approaches for reliability make the same assumptions. Indeed, if the network or endpoints fail permanently, recovery is impossible.

Two ways that information protocols can support tolerating permanent disruptions of selected agents are *history replay* and *role replacement* [14]. Because all of an agent's interactions are recorded in its local state, an agent can be restored from a crash by replicating its local and internal state and resuming work with the enabled messages that have not yet been sent. If an agent is completely unrecoverable or uncooperative, a replacement can be chosen using dynamic role binding.

Research on multiagent applications addresses fault tolerance [15]. However, existing works either focus on replicating agents [16] or adopt a distribution standpoint on how to reconnect nodes [17].



(a) Forwarding incoming.

(b) Forwarding outgoing.

(c) Ack.

(d) Probing.

(e) Probe propagation.

Figure 3: Protocol extensions summarized.

7. Conclusion

Relying on infrastructure alone is insufficient for fault tolerance in decentralized applications; application-level fault detection and resolution are necessary. Practitioners increasingly recognize the need for application-level mechanisms [18]; however, lacking adequate support in programming models, practitioners must handcraft such mechanisms separately for each application. No wonder then that relying on infrastructure for fault-tolerance is the dominant modus operandi in application development. Indeed, *service meshes* for microservices go further in the wrong direction since they seek to make fault handling and recovery invisible to the application.

Bungie makes application-level fault tolerance possible. Information protocols, which underpin

Bungie, provide for application-level awareness of faults and recovery from them. Bungie helps identify the main weaknesses of an application and provides simple and modular strategies to mitigate them, requiring no more infrastructure support than is available in UDP.

Acknowledgments

Thanks to NSF (grant IIS-1908374) and EPSRC (grant EP/N027965/1) for support.

REFERENCES

1. RosettaNet. Home page, 1998. www.rosettanet.org.
2. HL7. HL7 FHIR release 4. http://hl7.org/fhir/, Oct. 2019.
3. FpML. FpML 5.11 recommendation. https://www.fpml.org/spec/fpml-5-11-9-rec-1/, Dec. 2019.
4. J. H. Saltzer, D. P. Reed, and D. D. Clark. End-to-end arguments in system design. *ACM Transactions on Computer Systems*, 2(4):277–288, Nov. 1984. doi: 10.1145/357401.357402.
5. D. Clark. The network and the OS. In *SOSP History Day*, pages 11:1–11:19, Monterey, California, Oct. 2015. ACM. doi: 10.1145/2830903.2830912.
6. M. P. Singh. 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)*, pages 491–498, Taipei, May 2011. IFAAMAS. doi: 10.5555/2031678.2031687.
7. OMG. Business Process Model and Notation (BPMN), version 2.0.2, Jan. 2014. Object Management Group. https://www.omg.org/spec/BPMN/.
8. N. Yoshida, R. Hu, R. Neykova, and N. Ng. The Scribble protocol language. In *Trustworthy Global Computing - 8th International Symposium, TGC*, pages 22–41, 2013. doi: 10.1007/978-3-319-05119-2_3.
9. D. Ancona, A. Ferrando, and V. Mascardi. Parametric runtime verification of multiagent systems. In *Proceedings of the 16th Conference on Autonomous Agents and MultiAgent Systems, AAMAS*, pages 1457–1459, 2017.
10. A. K. Chopra, S. H. Christie V, and M. P. Singh. An evaluation of communication protocol languages for engineering multiagent systems. *Journal of Artificial Intelligence Research*, 69:1351–1393, 2020.
11. J. Shore. Fail fast. *IEEE Software*, 21(5):21–25, 2004. doi: 10.1109/MS.2004.1331296.
12. G. Candea and A. Fox. Crash-only software. In *Proceedings of HotOS'03: 9th Workshop on Hot Topics in Operating Systems*, pages 67–72. USENIX, 2003. URL https://www.usenix.org/conference/hotos-ix/crash-only-software.
13. J. Iyengar and M. Thomson. QUIC: A UDP-based multiplexed and secure transport. Technical report, Internet Engineering Task Force (IETF), Fremont, California, Oct. 2020. Proposed standard; https://datatracker.ietf.org/doc/draft-ietf-quic-transport/.
14. S. H. Christie V and A. K. Chopra. Fault tolerance in multiagent systems. In *International Workshop on Engineering Multi-Agent Systems*, pages 78–86. Springer, 2020.
15. K. Potiron, P. Taillibert, and A. E. Fallah-Seghrouchni. A step towards fault tolerance for multi-agent systems. In *Languages, Methodologies and Development Tools for Multi-Agent Systems, First International Workshop, LADS*, volume 5118 of *Lecture Notes in Computer Science*, pages 156–172. Springer, 2007. doi: 10.1007/978-3-540-85058-8_10.
16. Z. Guessoum, J. Briot, and N. Faci. Towards fault-tolerant massively multiagent systems. In *Massively Multi-Agent Systems I, First International Workshop, MMAS*, volume 3446 of *Lecture Notes in Computer Science*, pages 55–69. Springer, 2004. doi: 10.1007/11512073_5.
17. A. Ricci, A. Ciortea, S. Mayer, O. Boissier, R. H. Bordini, and J. F. Hübner. Engineering scalable distributed environments and organizations for MAS. In *Proceedings of the 18th International Conference on Autonomous Agents and MultiAgent Systems, AAMAS*, pages 790–798. IFAAMAS, 2019.
18. M. de Graauw. Nobody needs reliable messaging. https://www.infoq.com/articles/no-reliable-messaging/, June 2010.

Samuel H. Christie V is a Research Associate at

Lancaster University and a PhD student at NC State University. Contact him at schrist@ncsu.edu.

Amit K. Chopra is a Senior Lecturer at Lancaster University. Contact him at amit.chopra@lancaster.ac.uk.

Munindar P. Singh is a Professor in Computer Science at NC State University. Singh is a Fellow of IEEE, AAAI, and AAAS, and a former Editor-in-Chief of *IEEE Internet Computing* and *ACM Transactions on Internet Technology*. Contact him at singh@ncsu.edu.