Bruno: Protocol-based Garbage Collection and Information Management

Samuel H. Christie V[0000−0003−1341−0087],
Amit K. Chopra[0000−0003−4629−7594],
and Munindar P. Singh[0000−0003−3599−3893]

Lancaster University, Bailrigg, Lancaster, LA1 4YW, UK
{samuel.christie,amit.chopra}@lancaster.ac.uk
North Carolina State University, Raleigh, North Carolina, 27695, USA
{schrist,mpsingh}@ncsu.edu

Abstract. Business communications contain information of value to the parties involved. Some information represents and enables currently ongoing transactions. For purposes of efficiency, parties would want such information to be available in local storage, which is fast to access but possibly expensive per unit capacity. Other information is usually consigned to "offline" archival and may support a variety of purposes, e.g., analysis, reporting, meeting legal requirements, and so on. Archival storage is typically both significantly slower and less expensive compared to local storage. Businesses typically have policies based on which they consign information to archive. A related challenge is how to understand and process business events whose context has been archived.
We refer to the idea of archiving information as *garbage collection*. We present a multiagent conception of garbage collection based on protocols. Agents represent the involved parties and enact protocols. Protocol enactments represent transactions. An agent applies policies to decide which transactions to archive. Although the policies can be arbitrary, we show how to exploit the protocol to inform them. We also discuss how an agent may handle events whose context may be in archive.

1 Introduction

Modern business applications are increasingly decentralized and involve large amounts of communication and information. The data they generate can be ephemeral (e.g., a stateless web service could reply to a request and then discard it), but increasingly that information is preserved.

The primary reason is to support further computation. When a customer begins a purchase at an online store, the service must remember the contents of their cart throughout the interaction until the purchase is completed or canceled.

Secondary reasons for retaining data include logging for debugging purposes, and long term archival for regulatory compliance. Secondary uses for data are common enough, and long-term storage cheap enough, that the default is now to save everything indefinitely. For example, even AWS Lambda [2]—a light-weight

service for invoking invidividual functions—logs every invocation (the request, runtime logging, and results) indefinitely by default.

Retaining this data has a cost. Even database storage, which is much cheaper than memory but still easy to query using indexes and map/reduce operations, has capacity limits and costs more than longer term archives. According to Oracle [7], 62% of companies are using primary storage for inactive data, 80% of data goes unused after 90 days, and long-term storage (such as tape archives) can be less than 1/20th the price of active storage. Thus, most systems (such as AWS CloudWatch [1], which manages logging data) support *retention policies*, usually based on the number of days to keep a particular record. When the record expires, it is automatically deleted or moved to an archive.

However, these retention policies are usually statically configured at the infrastructure level. Although cloud services provide virtual infrastructure that can be reconfigured for each application, it still exists in a separate layer of the architecture intentionally kept opaque from application concerns and information. This means the application cannot make data retention decisions based on runtime information, such as who is making the request (priority customers could have their data stored longer, or have faster speeds), or based on the request parameters. This mismatch between application needs and infrastructure capabilities evokes the *end-to-end* principle[8]:

> [Some functions] can completely and correctly be implemented only with the knowledge and help of the application standing at the end points of the communication system. Therefore, providing that questioned function as a feature of the communication system itself is not possible. (Sometimes an incomplete version of the function provided by the communication system may be useful as a performance enhancement.)

Applied to our domain, data management policies implemented in the infrastructure may be employed as a performance enhancement, but may not be capable of completely and correctly matching the business requirements.

Garbage collecting and archiving interaction information is particularly challenging, because that information is essential for agent decision-making and correctness. If the information is archived improperly and the interaction continues, the agent could violate integrity. Thus, interaction information management is an application-level concern that must take the protocol specification into account to ensure correctness.

2 Running Example

To illustrate some of the challenges of data management, we specify a hypothetical e-commerce scenario.

Our scenario is a purchase protocol where the buyer can request a quote for an item and the seller replies with a time-limited offer. If the offer expires or the

purchase completes, then the seller can archive it to long-term storage without needing to restore it again.

Here is a detailed list of application requirements:

R0: The interaction involves two roles: CUSTOMER and SELLER
R1: CUSTOMER can request a quote from SELLER based on a query, along with a UUID to identify the interaction
R2: SELLER can give CUSTOMER an offer for the item at a specific price, along with an expiration date for the offer. Until the offer expires, SELLER guarantees the item is available at that price.
R3: SELLER may alternatively reject the request for quote with an explanation; e.g., it is receiving too many requests, or the item is unavailable.
R4: CUSTOMER can accept an offer before it expires. To accept the offer, CUSTOMER must reference the quote and its terms (price, expiration date, etc.).
R5: SELLER should confirm the acceptance if it arrives before expiration. Confirmation completes the interaction, and SELLER may then archive the transaction.
R6: However, if CUSTOMER attempts to accept an expired offer, SELLER should send a rejection message indicating that the offer is expired without restoring the interaction.

2.1 Bruno, our proposed solution

With Bruno, named after Bruno the Trashman from Sesame Street, we propose that data retention decesions be made at the application level using business requirements and possibly runtime data. Further, we propose that using information protocols to model the communication within a system can help identify at a fine grain which information can be safely archived, and help restore data for further interactions if necessary.

Listing 1. The Expiring Offer protocol

```
Expiring Offer {
  roles Customer, Seller
  parameters out oID key, out item, out price, out done
  private query, expiration, acceptance, confirmation, rejection

  Customer -> Seller: RFQ[out oID key, out query]
  Seller -> Customer: Offer[in oID key, in query, out item, out price, out
      expiration]
  Customer -> Seller: Accept[in oID key, in price, in expiration, out
      acceptance]
  Seller -> Customer: Confirm[in oID key, in acceptance, out confirmation,
      out done]
  Seller -> Customer: Reject[in oID key, in acceptance, out rejection, out
      done]
}
```

3 Background: Information Protocols

We formalize the above scenario using BSPL [9] with the *Expiring Offer* protocol given in Listing 1.

In *Expiring Offer*, there are two roles, CUSTOMER and SELLER; when the protocol is enacted each role is played by an agent. The public parameters are oID, item, price, and done—together these parameters constitute the public interface of the protocol for composition. Enactments are identified by their keys, in this case the single parameter oID. For any given binding of the key parameters, the other parameters may have at most one binding; as such, parameter bindings in information protocols are monotic and immutable.

Three aspects of information protocols are particularly important for our data management objectives:

3.1 Information dependencies

The parameter adornments identify information dependencies between messages. If a schema contains a parameter adorned ⌜in⌝, then it may not be sent until a binding for that parameter is observed by the agent. Parameters adorned ⌜out⌝ introduce new bindings (resulting in their observation by both the sender and recipient), and ⌜in⌝ parameters copy information already known by the sender to the recipient.

For example, in *Expiring Offer* the *Accept* message has acceptance as ⌜out⌝, and oID, price, and expiration as ⌜in⌝. Thus, it cannot be sent until those ⌜in⌝ parameters are all bound: oID by *RFQ*, and price and expiration by *Offer*.

Dependency-based enablement provides flexibility and avoids explicit temporal ordering. Further, these dependencies can identify which information is still useful and which can be garbage collected.

3.2 Integrity constraints

The dependencies and single-binding-per-enactment constraint are referred to as the *integrity constraints* for protocol enactments. In current implementations, message (emission or reception) that violates these constraints is discarded, which is why careful management of enactment information is essential for correctness.

If information is evicted before an enactment is complete, it is possible that a reception will appear to violate integrity simply because its enactment is forgotten, creating an opportunity for new policies that handle integrity violation.

3.3 Protocol completion

In addition to specifying the public interface for composition, public parameters also define the extent of the protocol enactments: when all of the parameters are bound, the enactment is *complete*. Any further messages transmitted after the public parameters are bound would have no visible effect.

This concept is important for garbage collection or archival: when an enactment completes, it will not be extended further, and no more decisions will need to be made with its information, so the enactment can be archived.

4 Protocol-Based Solution

We now give an overview of our proposed solution, which uses information from the protocol specification to enable application-level data management policies

4.1 Architecture

Figure 1 shows our proposed agent architecture with special focus on components added specifically for supporting data management, which are framed in red.

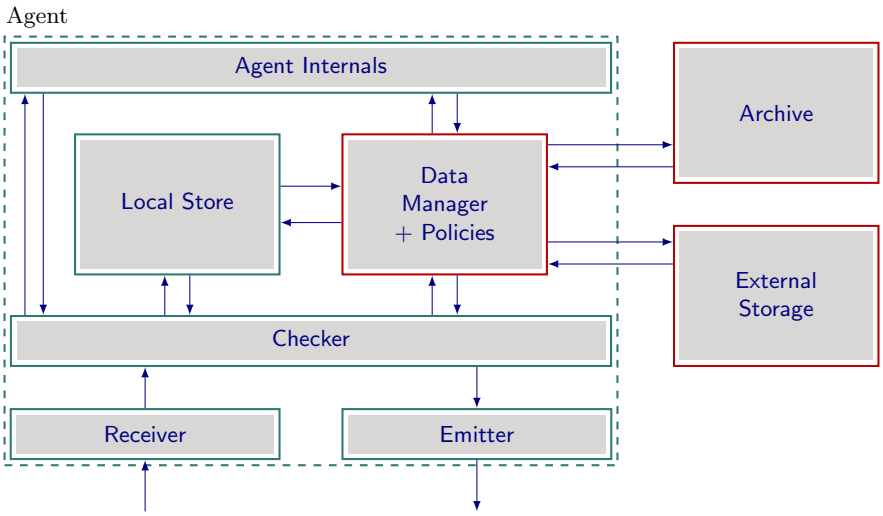

Fig. 1. Bruno architecture.

The unique points of this architecture are the Data Management Policies, and separation of the agent's storage into three categories.

The Local Store has long been a component of information protocol agent architectures. It represents the communications that the agent has observed, both incoming and outgoing. Effectively, the Local Store logs the history of observed messages.

The Local Store is responsible for the integrity of the agent, and therefore must maintain the desired level of consistency. This usually means that the Local Store involves some amount of persistence, so that the agent can resume operation after a crash without losing information. Otherwise, if the agent were to be resurrected without its history, it may send messages that are inconsistent with other agents' histories. For example, if the SELLER were to crash after sending *Confirm* and lose the message, it might be resurrected in a state where

it believes that enactment is incomplete, and attempt to complete it with the *Reject* message instead.

Thus the Local Store is responsible for providing the level of consistency required by the system. In some applications, the requirements may actually be very low. For example, if the agent is stateless or deterministic based on its inputs, it need not store any history. Or, if the agent is not expected to be resurrected after a crash, it could be implemented using a memory-only local store.

The Archive represents permanent storage; once archived, enactment data may be accessed infrequently for reporting or auditing purposes, but the enactment itself will not continue.

This component could be implemented using e.g. the AWS Glacier Deep Archive storage class [3], which takes 12 to 48 hours to retrieve information, and is expected to be accessed once or twice per year.

The External Storage encompasses anything that is not in the usable Local Store, yet is not in permanent Archive and so may be restored for further use.

External storage could be implemented by other agents offering storage services, and a single agent could have policies that direct different types of information to different stores with different performance characteristics.

The Data Manager (DM) ties all of these storage categories together, making dynamic decisions about where information should be located.

First, the DM is initialized and configured by the Agent Internals.

The DM interacts with the Checker to inform it regarding integrity and expiration; if a message belongs to an enactment that has expired, it can be quickly rejected without reference to other storage.

The DM interacts with the Local Store to offload and reload information according to caching policies and resource limits.

Finally, the DM uses the Archive and External Storage to offload information from the Local Store.

4.2 Example Data Management Policies

Now that our architecture and the interactions between the various components have been explained, we will give some examples of policies that the agents could employ to manage their data.

Our policy examples are given in a simple YAML-based [10] configuration format. Each policy is an element in a list, so a given protocol may have an arbitrary number of policies. The policies are specified as dictionaries, with their key/value pairs describing various options and parameters.

Parameterized expiration Where standard infrastructure-level retention policies give a fixed lifetime to broad categories of data (e.g. one month after creation), *parameterized* expiration takes advantage of information available to the application at runtime, specifically interaction parameter values. Parameterized expiration is also useful because it can enable context-free triage of subsequent messages.

For example, in *Expiring Offer*, the *Offer* message includes the expiration date as a parameter. It may be that different offers have different pricing cycles, or that the cycle is based on business hours rather than the event itself. The resulting policy would be something like Listing 2, which specifies that enactments of *Expiring Offer* expire after the date specified in its expiration parameter.

Listing 2. Parameterized Expiration Policy

```
Expiring Offer:
- expire: enactment
  after: expiration // this is an interaction parameter
```

Including the expiration information as a parameter in the message, and then requiring it in subsequent messages from CUSTOMER significantly simplifies triage of messages belonging to missing (and thus potentially archived) enactments. That is, CUSTOMER's *Accept* must include the expiration date, and thus proves itself to be invalid after that date without reference to any other data. Note that to ensure the validity of the parameter, it must be signed by SELLER, and include the enactment keys as input to the signature; otherwise CUSTOMER could either fabricate an expiration date, or reuse an expiration date from another enactment in a replay attack.

Archiving complete enactments When an enactment is complete, there should be no further messages and it can be safely archived. This policy can be automatically computed from the protocol specification.

Listing 3. Archiving upon completion

```
Expiring Offer:
- archive: enactment
  when: complete
```

Based on the specification of *Expiring Offer*, completion occurs when oID,item,price and done are all bound, which should occur when either *Confirm* or *Reject* is sent. Unfortunately, this approach to archival is not self-attesting, and therefore may introduce ambiguities in the local store. To avoid that, the DM could replace any prematurely archived enactment with a small *tombstone* marker that indicates which store it has been relocated to and why. This way, if CUSTOMER attempts to resend the *Accept* message after SELLER thought it was complete (perhaps because the *Accept* or *Reject* was lost in transit), the *Seller* can respond accordingly without necessarily reloading the entire enactment from the archive, or having to search for the enactment across multiple external stores. The easiest way to handle it is just to ignore the message, but an optional continuation strategy could be specified.

Listing 4. Archiving upon completion with tombstone

```
Expiring Offer:
- archive: enactment
  when: complete
  tombstone: True
  continuation: ignore
```

Evicting and restoring least-recently-used enactments More generally, the agent could evict information from the Local Store at any time, though it may need to restore the information if the enactment continues.

Eviction policies have three primary parameters: *when* to evict (the trigger), *how much* to evict (target), and *which* information to evict (criteria).

Possible triggers include memory or database usage exceeding a threshhold, or a daily garbage collection batch process. The target should generally be related to the trigger, but should provide a safety margin so that the eviction policy does not repeatedly activate as new messages come in. E.g. setting a trigger when memory reaches 90% might evict one enactment when the memory usage crosses that threshold, but without a target it would be triggered again when another message arrives.

The eviction criteria should be a prioritization function, so that the policy can continue evicting items until its target is reached. A reasonable criteria for eviction might be the classic Least Recently Used algorithm, which would evict enactments that have not recently observed any messages. In many cases (though obviously not all), the probability of a message arriving is inversely proportional to the time since the last one was observed; this naturally follows from the fact that when an agent abandons an enactment they won't send any more messages.

Listing 5. LRU eviction policy

```
Expiring Offer:
- evict: enactments
  when: memory > 90%
  target; memory < 70%
  criteria: now - updated
  sort: >
```

Eviction policies should maintain a priority queue of items to evict, and once triggered repeatedly evict the highest priority items until the target is reached.

Another potentially useful policy is to evict the enactments closest to their expiration dates. Though in some cases (such as conference submission deadlines) procrastination may make these enactments *more* likely to see further action, in many cases the closer to expiration the less likely they are to complete.

Listing 6. Impending expiration eviction policy

```
Expiring Offer:
- evict: enactments
  when: memory > 90%
  target: memory < 70%
  criteria: expiration - now
  sort: <
```

Garbage collecting unnecessary parameters With application-level policies, eviction may be applied at a finer grain than complete enactments. For example, in *Expiring Offer*, the query parameter is no longer required after *Offer* has been sent—no subsequent messages reference this parameter. Thus, we can create a policy to evict that parameter from the local store once it is no longer needed.

Listing 7. Evict unnecessary parameters

```
Expiring Offer:
- evict: parameters
  when: unnecessary
```

The above policy simply evicts all parameters when they become unnecessary, according to the protocol dependencies. Thus, care should be taken that the protocol properly reflects the dependencies of the decisions made at each point. However, more precise policies could be written:

Listing 8. Evict unnecessary parameters

```
Expiring Offer:
- evict: query
  when: sent Offer
```

This policy clearly identifies a single triggering event, so it can be implemented efficiently by listening only for emission of the *Offer* message.

External and Multimedia Data Most parameters are assumed to have semantic value for the interaction; that is, the parameter bindings have meaning to the agents that they may use to make future decisions. Thus, that information needs to be kept around in the local store so the agent can access it when making a relevant decision. However, in some cases, such as transferring multimedia, the data is opaque and not used directly for decision-making.

In these cases, we can offload the parameters to external storage immediately. For example, suppose we have a *Multimedia* protocol, in which some message contains a video parameter, which is only used to transfer video data that is not interpreted directly by the agents. Then we can use a policy like the following to replace the parameter contents with a reference to its storage location:

Listing 9. Evict unnecessary parameters

```
Multimedia
- external: video
```

This policy replaces the video data with a reference to an external storage location. We consider external and multimedia storage further in Section 5.

More Sophisticated Policies The policy examples given above are fairly simple, in that they can be declaratively specified using only a few domain-specific terms (such as whether to evict enactments or parameters, or targetting memory usage). However, it is quite certain that the few concepts and criteria that we imagined during the writing of this paper will not encompass every business case. For example, suppose the seller wanted to use a predictive model

based on past customer behavior. Capturing such a model is definitely beyond the scope of our simple policy language, so we offer hybrid extensibility instead.

In the simple case, a policy designer can use the normal DSL, and include a function invocation to delegate the decision to e.g. Python code.

Listing 10. Policy Delegation

```
Expiring Offer:
- evict: enactments
  when: predict_abandonment(enactment)
  check: daily
```

This example policy invokes a `predict_abandonment` function, passing the enactment object as an argument. The status of the predictive model is checked daily, and any enactments for which the model returns True are evicted.

For even more complex cases, the policies could be managed directly via an API. That way, the developers have full control over when the policies are invoked, and how they are executed. Importantly, direct API access would enable more efficient complex interactions with external resources, such as databases or HTTP endpoints.

4.3 Integrity violation policies

Finally, we consider the case of received messages apparently violating integrity. In past implementations of information protocol agent development frameworks, such invalid messages were simply discarded. However, now we have an opportunity to consider the problem with more nuance, and provide more mechanisms for resolution.

The apparent integrity violation could be either because the incoming message conflicts with the information in the Local Store for the matching enactment, or because it references but does not initiate an enactment that is not present in the Local Store.

Depending on the details, there are several possibilities for why an incoming message may appear to violate integrity:

1. The receiving agent is at fault, because it lost the context:
   e.g. the enactment was archived
2. The sending agent is at fault:
   (a) Ignorance: the sending agent lost their context
      e.g. because it was resurrected incorrectly and lost part of its history
   (b) Incompetence: the sending agent is implemented incorrectly
   (c) Malice: the sending agent is intentionally sending incorrect messages

Unfortunately, information management alone is likely inadequate for addressing malice or incompetence, so we defer their consideration. The two cases caused by loss of context, however, represent potentially reconcilable disagreements about interaction information.

(Receiver) Missing Context If the recipient does not have the context, then it may have been archived and thus difficult to restore. Certainly, if the enactment is completely missing from the Local Store (there is no tombstone), then it may be unclear where the data might have moved to. Thus, in this case the agent could take one of several actions:

1. Discard the message
2. Log and escalate the error
3. Request more information about the enactment from the sender

Requesting more information about the enactment may be a special case of retrieving information from External Storage. If the other agent implements the External Storage protocol, and the parameters are properly signed (e.g. to prevent the customer from fabricating the offer price), then the information can be retreived and restored safely.

(Sender) Disagreement with Context If the data is merely incorrect, and we assume the sender is neither malicious nor faulty, then it may be the result of lost history. The list of actions the agent can take therefore is similar:

1. Discard the message
2. Log and escalate the error
3. Engage in consensus protocol with other agent

In this case, because the information was available but contradictory, merely providing the original enactment data may not be enough to restore consistency. Of course, the consensus protocol need not be complicated, and may only require transfer and acceptance.

5 Discussion

Garbage collection and data management do not seem to be popular topics for multiagent systems researchers, although there are many papers on agents operating under resource constraints (budget, communication, information, etc.) Wright et al. [12] examined multiagent system organization and its affect on data management; too many relationships limit scalability. Uchiya et al. [11] built a multiagent-system-based backup tool, which might in some respects be useful for building an External Storage component, but is otherwise not concerned with offloading and reloading information based on application needs. Conway et al. [5] describe a policy-based data management system, but focus on classic archival and long-term storage, not active interaction information.

The approach we have proposed, of application-level data management policies based on information protocol specifications, appears to have many advantages over existing solutions that either ignore storage constraints or set static retention policies.

These policies are very flexible, since they can take into account information only availble at runtime through inspecting message parameters. They can be

more aggressive, evicting data as soon as the interaction is complete instead of waiting for a longer expiration period. Involving the protocol specification enables propagation of expiration details, so that messages can be triaged without referring to any other data. When evicted, the enactment keys enable the use of tombstones identifying the present status and location of the offloaded information, opening the possibility for decentralized multiagent storage management instead of a single well-known storage service. Finally, the transparent structure of messages schemas enables very fine-grained data management policies, which can evict parts of enactments once they are no longer needed.

Comparison With Other Work on Policies Since our work focuses on policies for information management, it is also fair to ask how our concept compares to other work on policies, including policy languages [6] or agent reasoning frameworks such as JaCaMo [4].

In contrast to these policy-focused works, our focus is on enabling application-level management of information, specifically interaction information with the challenge of preserving correctness. We have illustrated our ideas with policy sketches to show examples of what might be possible with application-level information management and how a solution can utilize protocol specifications, but our focus is not the policy specification itself. In fact, it might be a good direction for further work to use an existing policy language or reasoning framework to implement the Bruno architecture.

Tradeoffs of Application-Level Management A common perception is that moving features from the infrastructure level to the application level trades efficiency for flexibility. For example, AWS does not appear to charge fees for their retention policies, whereas implementing the same policies in userspace would require paying both for computation and database requests to access and then delete the information. Another potential tradeoff is in complexity: developers can treat infrastructure-level solutions as black boxes, but must implement application-level solutions themselves.

Having not yet implemented the system, we unfortunately cannot give a performance evaluation to estimate the magnitude of these tradeoffs.

However, these tradeoffs may prove not to be significant for several reasons: First, the choice to use application-level policies is not all-or-nothing; policies can be incrementally added on top of existing infrastructure-level solutions. Second, complexity can be managed through the use of libraries, tools, and appropriate defaults; our policy examples are an attempt to sketch what such an approach might look like, and show how it could be easy to add even sophisticated policies to a system. Finally, efficiency is never worth sacrificing correctness; because infrastructure policies are not aware of application requirements, they will necessarily be conservative, and so leave some room for application-level policies to improve performance without sacrificing correctness (assuming they can be implemented efficiently).

Also, many platforms do not really offer alternative solutions anyway. AWS does offer retention policies for some of its services, but many databases and application frameworks do not have such features.

Multimedia and Opaque Attachments Another application is the managment of large, opaque attachments. For example, if an agent participates in a file sharing or video streaming protocol: the video segments included in each message are not really individually meaningful, but would take up immense amounts of space in the Local Store or history. One possible solution is to apply the tombstone concept at the parameter rather than enactment level: as each piece is received, store it in the appropriate final destination, and avoid duplicating storage in the communication history by replacing the data with pointers to its new location. That way the communication history takes up very little space, yet none of the meaning is lost. There is a risk, however, if the data on disk changes or is deleted without updating the reference. Thus, a checksum or hash should be used as part of the reference to prevent history fabrication.

External Storage The concept of External Storage is also a very open subject. On one end, the concept of External Storage encompasses any copies of the communication history that are not directly part of the Local Store (and therefore are not directly usable in integrity checking). This storage could be driven directly by the Data Manager, and automatically queried whenever information is needed but not present in the Local Store. On the other hand, External Storage could be implemented as a multiagent system itself, with agents renting storage space to each other, or replicating information redundantly throughout the same multiagent system. The storage market case would require the Data Manager and its policies to negotiate for, procure, and then manage storage across a widely varied set of service providers, each with their own reliability and cost. It is also somewhat challenging to think that an agent may be engaging in communication for the purpose of managing its communication data. In the simpler MAS replication case, the agent may intentionally replicate more information than necessary to its peers, so that it can recover the data later if necessary; effectively using the counterparties it is already interacting with as backups instead of involving unrelated parties. Conversely, the agents could communicate only what the protocols indicate, but then request copies of the relevant messages when necessary. In this case, so long as only one agent crashes at a time (and all messages arrive) there will be no data lost: it simply requests all of the mutually shared history from each of the roles it interacts with, effectively re-enacting the history.

Generalization and Application for Existing Systems Because data management is an important problem in modern software systems even beyond MAS and interaction protocols, a fair question is how this work might be applied to existing or mainstream applications.

The essential aspects provided by interaction protocol specifications are the concept of enactment completion, and the explicit information dependencies which help identify when a parameter is no longer useful. These concepts help identify which information can be garbage collected or archived without compromising correctness. Although it is possible that a protocol specification could be written for an existing application, we suspect that most systems are not designed to meet the rigorous constraints (such as the key and integrity constraints) that BSPL protocols require.

However, the rest of the solution—and especially the concept of application-level information management—should be transferrable to an existing application. Because the information being managed—message instances and parameters—correspond to rows and columns in a database respectively, it should not be hard to implement an equivalent Data Manager component for working with any database system. Without messaging, some other event source would be needed to trigger the policies.

6 Conclusion

The end-to-end principle suggests that although some features in a system may be more efficiently implemented in the supporting infrastructure, they should still be pushed to the application level where they would be re-implemented for correctness anyway. Data management is essential to the correctness of most complex multiagent systems, because they depend on that data for decision-making. So long as assumptions of infinite storage hold, there is no need for any data management; but when resources become constrained, data must be managed or lost, and then it is best managed by the application where the best and most informed decisions can be made. Information protocols further provide support for data management decisions at the application level.

We have only begun to scratch the surface of how information protocols could be applied to the problem of data management, and how a Data Manager and its policies can be implemented. In future work, we hope to actually implement such a system, more thoroughly explore the space of potential policies, and perform some experiments to identify which policies are likely to be most useful in practice.

Acknowledgments. Christie and Chopra were supported by EPSRC grant EP/N027965/1.

Bibliography

[1] Amazon Web Services: AWS Cloudwatch (Mar 2022), https://aws.amazon.com/cloudwatch/, accessed 2022-03-11

[2] Amazon Web Services: AWS Lambda (Mar 2022), https://aws.amazon.com/lambda/, accessed 2022-03-11

[3] Amazon Web Services: S3 Storage Classes (Mar 2022), https://aws.amazon.com/s3/storage-classes/glacier/, accessed 2022-03-11

[4] 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 (Jun 2013). https://doi.org/10.1016/j.scico.2011.10.004

[5] Conway, M., Moore, R., Rajasekar, A., Nief, J.Y.: Demonstration of policy-guided data preservation using iRODS. In: Proceedings of the IEEE International Symposium on Policies for Distributed Systems and Networks (POLICY). pp. 173–174. IEEE Computer Society, Pisa (Jun 2011)

[6] Damianou, N., Dulay, N., Lupu, E., Sloman, M.: The Ponder policy specification language. In: Proceedings of the International Workshop on Policies for Distributed Systems and Networks (POLICY). pp. 18–38. No. 1995 in Lecture Notes in Computer Science, Springer, Bristol, United Kingdom (Jan 2001)

[7] Oracle: What are the benefits of data archiving with oracle? (Jul 2015), https://www.oracle.com/assets/why-archive-with-oracle-article-seo-2608143.pdf, accessed 2022-03-11

[8] Saltzer, J.H., Reed, D.P., Clark, D.D.: End-to-end arguments in system design. ACM Transactions on Computer Systems **2**(4), 277–288 (Nov 1984). https://doi.org/10.1145/357401.357402

[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 (AAMAS). pp. 491–498. IFAAMAS, Taipei (May 2011). https://doi.org/10.5555/2031678.2031687

[10] Team, Y.L.D.: YAML. https://yaml.org/spec/1.2.2/ (Apr 2022)

[11] Uchiya, T., Shibakawa, M., Takumi, I., Kinoshita, T.: Multiagent-based distributed backup system for individuals. In: 14th IEEE/ACIS International Conference on Computer and Information Science, ICIS. pp. 361–366. IEEE Computer Society (2015). https://doi.org/10.1109/ICIS.2015.7166620

[12] Wright, W., Moore, D., Thome, M.: Organizational patterns for data management in large-scale distributed multiagent systems. In: The Second International Joint Conference on Autonomous Agents & Multiagent Systems, AAMAS July 14-18, 2003. pp. 1162–1163. ACM (2003). https://doi.org/10.1145/860575.860846