A Linguistic Framework for Dynamic Composition of Dependability Protocols[*]

Gul Agha, Svend Frølund, Rajendra Panwar and Daniel Sturman

Department of Computer Science
1304 W. Springfield Avenue
University of Illinois at Urbana-Champaign
Urbana, IL 61801, USA
Email: {agha | frolund | panwar | sturman}@cs.uiuc.edu

Keywords: Programming Languages, Composition of Protocols, Dependable Systems, Software Fault-Tolerance

Abstract

We present a language framework for describing dependable systems which emphasizes *modularity* and *composition*. Dependability and functionality aspects of an application may be described separately providing a separation of design concerns. Futhermore, the dependability protocols of an application may be constructed bottom-up as simple protocols that are composed into more complex protocols. Composition makes it easier to reason about dependability and supports the construction of general reusable dependability schemes. A significant aspect of our language framework is that dependability protocols may be loaded into a running application and installed dynamically. Dynamic installation makes it possible to impose additional dependability protocols on a server as clients with new dependability demands are integrated into a system. Similarly, if a given dependability protocol is only necessary during some critical phase of execution, it may be installed during that period only.

1 Introduction

This paper describes a software methodology for supporting dependable *services* in *open systems*. Provision of a dependable service involves the *servers* implementing the service as well as the communication channel to the servers. We make no assumptions about the behavior of the customers, called *clients*, using the service. For our purposes, the most significant characteristic of an open

[*]The research described has been made possible by support provided by a Young Investigator Award from the Office of Naval Research (ONR contract number N00014-90-J-1899) and by an Incentives for Excellence Award from the Digital Equipment Corporation Faculty Program. The authors would like to thank Chris Callsen, Wooyoung Kim, and Anna Patterson for helpful discussions concerning the manuscript. We would also like to thank Takuo Watanabe for his insights into the use of reflection and Chris Houck for the modifications to the HAL compiler required for our implementations.

system is *extensibility*: new services and new clients may be integrated into an open system while it is functioning.

In many existing methodologies for programming dependable applications, the dependability characteristics of an application are fixed statically (i.e., at compile time). This is unsatisfactory in many computer systems, which are required to function for a long period of time, yet are fault-prone due to the uncertain environment in which they operate. An example of such a system is the control system embedded in an orbiting satellite. Furthermore, in open systems the addition of new services and clients may impose new requirements for dependability of a service. For example, a file server may start only addressing safety by checkpointing the files to stable storage. In an open system, new clients added to the system may require the server to also provide security, e.g., by encrypting the files they transfer to the clients. Our method includes *dynamic installation* of dependability protocols which allows a system to start with a "minimal" set of dependability protocols and later be extended with more protocols where and when the need arises. As the file server example illustrates, not all dependability protocols that may be needed at runtime can necessarily be predicted at compile time.

Our methodology incorporates object-oriented programming methods and as a result offers the following advantages:

- *Separation of design concerns*: an application programmer need not be concerned with the particular dependability protocols to be used when developing an application.

- *Reusability*: code for implementing dependability protocols and application programs can be stored in separate libraries and reused.

We employ *reflection* as the enabling technology for dynamic installation of dependability protocols. Reflection means that a system can reason about, and manipulate, a representation of its own behavior. This representation is called the system's *meta-level*. In our case, the meta-level contains a description of the dependability characteristics of an executing application; reflection thus allows dynamic changes in the execution of an application with respect to dependability. Reflection in an object based system allows customization of the underlying system independently for every object as compared to customization in a micro kernel based system [ABB$^+$86] where changes made to the micro kernel affect all the objects collectively. This flexibility is required for implementing dependability protocols since such protocols are mostly installed on very specific subsets of the objects in a system.

The code providing dependability is specified independently from the code which specifies the application specific functionality of a system. As we show later, our reflective model allows *compositionality* of dependability protocols. Compositionality means that we can specify and reason about a complex dependability scheme in terms of its constituents. Thus, logically distinct aspects of a dependability scheme may be described separately. Compositionality supports a methodology in which dependability protocols are constructed in terms of general, reusable, components. Dynamic composition is particularly useful; it allows software for additional dependability protocols to be constructed and installed without knowledge of previously installed protocols. It may not be possible to describe a protocol in general terms. In such cases, composition may not be possible. For example, the composition of the two-phase commit protocol with security mechanisms may not be done naively [JM92].

A number of languages and systems offer support for constructing fault tolerant systems. In Argus [LS82], Avalon [DHW88] and Arjuna [PS88], the concept of nested transactions is used to structure distributed systems. Consistency and resilience is ensured by atomic actions whose effect are checkpointed at commit time. The focus in [MPS91], [Coo90] and [BJ87] is to provide a set of protocols that represent common communication patterns found in fault tolerant systems. None of the above systems support the factorization of fault tolerance characteristics from the application specific code. In [WL88] and [OOW91], replication can be described separate from the service being replicated. Our approach is more flexible since fault tolerance schemes are not only described separately but they can also be attached and detached dynamically. Another unique aspect of our approach is that different fault tolerance schemes may be composed in a modular fashion. For example, checkpointing may be composed with replication without having either method know about the other.

Reflection has been used to address a number of issues in concurrent systems. For example, the scheduling problem of the Time Warp algorithm for parallel discrete event simulation is modeled by means of reflection in [Yon90]. A reflective implementation of object migration is reported in [WY90]. Reflection has been used in the Muse Operating System [YMFT91] for dynamically modifying the system behaviour. Reflective frameworks for the Actor languages MERING IV and Rosette have been proposed in [FB88] and [TS89], respectively. In MERING IV, programs may access *meta-instances* to modify an object or *meta-classes* to change a class definition. In Rosette, the meta-level is described in terms of three components: a *container*, which represents the acquaintances and script; a *processor*, which acts as the scheduler for the actor; and a *mailbox*, which handles message reception.

The paper is organized as follows. Section 2 gives a more detailed description of the concept of reflection. Section 3 introduces the Actor model which provides a linguistic framework for describing our methodology. Note that our methodology is not dependent on any specific language framework; we simply use an Actor language as a convenient tool to describe our examples. Section 4 describes our *meta-level architecture for ultradependability* (MAUD). Section 5 gives an example of a replicated service implemented using MAUD. Composition of methods for dependability is addressed in Section 6 and illustrated in section 7 by means of an example composing replication with a two-phase commit protocol. The final section summarizes our conclusions and research directions.

2 Reflection

Reflection means that a system can manipulate a causally connected description of itself [Smi82, Mae87]. Causal connection implies that changes to the description have an immediate effect on the described object. The causally connected description is called a *meta-level*. In a reflective system, implementation of objects may be customized within the programming language. The customization can take place on a per *object* basis in the form of *meta-objects*. In this paper, we use the term *object* as a generic term for clients and servers (when it is not necessary to distinguish between them). The object for which the meta-object represents certain aspects of the implementation is called the *base object*. A meta-object may be thought of as an object that logically belongs in the underlying runtime system. A meta-object might control the message lookup scheme that would map incoming messages to operations in the base object. Using reflection, such implementation level objects can be accessed and examined, and user defined meta-objects may be installed, yielding a potentially customizable runtime system within a single language framework.

The reflective capabilities which are provided by a language are referred to as the *meta-level architecture* of the language. The meta-level architecture may provide variable levels of sophistication, depending on the desirable level of customization. The most general meta-level architecture is comprised of complete interpreters, thus allowing customization of all aspects of the implementation of objects. In practice, this generality is not always needed and, furthermore, defining a more restrictive meta-level architecture may allow reflection to be realized in a compiled language. The choice of a meta-level architecture is part of the language design. Customizability of a language implementation must be anticipated when designing the runtime structure. Although a restrictive meta-level architecture limits flexibility, it provides greater safety and structure. If all aspects of the implementation were mutable, an entirely new semantics for the language could be defined at runtime; in this case, reasoning about the behavior of a program would be impossible.

We limit our meta-level to only contain the aspects that are relevant to dependability. Application specific functionality is described in the form of base objects and dependability protocols are described in terms of meta-objects. Thus, dependability is modeled as a special way of implementing the application in question. Our methodology gives modularity since functionality and dependability are described in separate objects. Since meta-objects can be defined and installed dynamically, a system can dynamically switch between different dependability modes of execution. Dependability protocols could be installed and removed dynamically, depending on system need. Furthermore, new dependability protocols may be defined while a system is running and put into effect without stopping and recompiling the system. For example, if a communication line within a system shows potential for unacceptable error rates, more dependable communication protocols may be installed without stopping and recompiling the entire system.

Since meta-objects are themselves objects, they can also have meta-objects associated with them, giving customizable implementation of meta-objects. In this way, meta-objects realizing a given dependability protocol may again be subject to another dependability protocol. This scenario implies a hierarchy of meta-objects where each meta-object contributes a part of the dependability characteristics for the application in question. Each meta-object may be defined separately and composed with other meta-objects in a layered structure supporting reuse and incremental construction of dependability protocols.

Because installation of a malfunctioning meta-level may compromise the dependability of the a system, precautions must be taken to protect against erroneous or malicious meta-objects. To provide the needed protection of the meta-level, we introduce the concept of privileged objects called a *managers*. Only managers may install meta-objects. Using operating system terminology, a manager should be thought of as a privileged process that has *capabilities* to dynamically load new modules (meta-objects) into the kernel (meta-level). It should be observed that, because of the close resemblance to the operating system world, many of the operating system protection strategies can be reused in our design. We will not discuss particular mechanisms for enforcing the protection provided by the managers in further detail here. Because only managers may install meta-objects, special requirements can be enforced by the managers on the structure of objects which may be installed as meta-objects. For example, managers may only allow installation of meta-objects instantiated from special verified and trusted libraries. Greater or fewer restrictions may be imposed on the meta-level depending on the dependability and security requirements that a given system must meet.

3 The Actor Model

We illustrate our approach using the *Actor model* [Agh86, Agh90]. It is important to note that the idea of using reflection to describe dependability is not tied to any specific language framework. Our methodology assumes only that resources can be created dynamically, if needed to implement a particular protocol and that the communication topology of a system is reconfigurable. Our methodology does not depend on any specific communication model. In particular, it is immaterial to our approach whether synchronous or asynchronous communication is used.

Actors can be thought of as an abstract representation for multicomputer architectures. An actor is an encapsulated entity that has a local state. The state of an actor can only be manipulated through a set of *operations*. Actors communicate by asynchronous point to point message passing. A message is a request for invocation of an operation in the target actor. Messages sent to an actor are buffered in a *mail queue* until the actor is ready to process the message. Each actor has a system-wide unique identifier which is called a *mail-address*. This mail-address allows an actor to be referenced in a location transparent way. The *behavior* of an actor is the actions performed in response to a message. An actor's *acquaintances* are the mail addresses of known actors. An actor can only send messages to its acquaintances, which provides locality. In order to abstract over processor speeds and allow adaptive routing, preservation of message order is not guaranteed.

The language used for examples in this paper is HAL [HA92], an evolving high-level actor language which runs on a number of distributed execution platforms. HAL provides two other message passing constructs besides the asynchronous send. The first, `SSEND`, is a message order preserving send, or *sequenced send*. `SSEND` allows the sender to impose an arrival order on a series of messages sent to the same target. The second, `BSEND`, is a remote procedure call mechanism, or *blocking send*. The calling program implicitly blocks and waits for a value to be returned from the actor whose method was invoked.

4 A Meta-Level Architecture

In this section we introduce MAUD (**M**eta-level **A**rchitecture for **U**ltra **D**ependability). As previously mentioned, MAUD is designed to support the structures that are necessary to implement dependability. In MAUD, there are three meta-objects for each actor: *dispatcher*, *mail queue* and *acquaintances*. In the next three paragraphs we describe the structure of meta-objects in MAUD. Note that MAUD is a particular system developed for use with actors. It would be possible, however, to develop similar systems for most other models.

The dispatcher and mail queue meta-objects customize the communication primitives of the runtime system so that the interaction between objects can be adjusted for a variety of dependability characteristics. For instance, a dispatcher meta-object is a representation of the implementation of the `(SEND...)` action. Whenever the base object issues a message send, the runtime system calls the `transmit` method on the installed dispatcher. The dispatcher performs whatever actions are needed to send the given message. Installing new send behaviors makes it possible to implement customized message delivery patterns.

A mail queue meta-object represents the mail queue holding the incoming messages sent to an actor. A mail queue is an object with `get` and `put` operations. After installation of a mail queue meta-object, its `get` operation is called by the runtime system whenever base object is ready to process a message. The `put` operation on a mail queue is called by the runtime system whenever a message for the base object arrives. By installing a mail queue at the meta-level, it is possible to customize the way messages flow into the base object.

The acquaintances meta-object is a list representing the acquaintances of a base object. Information about an actor's acquaintances is necessary in order to checkpoint its state. For our current purposes, we assume that the acquaintances's meta-object is immutable. Otherwise, if customized acquaintance lists could be installed, static checking of legal names would be impossible.

Meta-objects are installed and examined by means of *meta-operations*. Meta-operations are defined in the class called `Object` which is the root of the inheritance hierarchy. All classes in the system inherits from `Object`, implying that meta-operations can be called on each actor in the system. The meta-operations `change-mailQueue` and `change-dispatcher` install mail queues and dispatchers for the object on which they are called. Similarly, the meta-operations `get-mailQueue`, `get-dispatcher` and `get-acquaintances` return the meta-objects of a given actor. If no meta-objects have been previously installed, an object representing the built-in, default, implementation is returned. Such default meta-objects are created in a lazy fashion when a meta-operation is actually called.

5 A Replicated Server

In a distributed system, an important service may be replicated to maintain availability despite processor faults. The type of faults that can be experienced are generally considered to be of two basic types: Byzantine failures or fail-stop failures [Sch90]. In this section, we will give an example of how MAUD can be used in an actor domain to implement a modular and application-independent protocol which uses *replication* to protect against fail-stop failures.

```
(DEFINE-ACTOR serveMailq
    (SLOTS data members)
    (METHOD (get who) )
    (METHOD (put msg)
       ;; A bcaster actor broadcasts msg to members
       (BSEND bcast (NEW bcaster msg) members)))
(DEFINE-ACTOR bcaster
    (SLOTS msg)
    (METHOD (bcast l)
       (IF (not (null? l))
          ((SSEND (msg-type msg) (car l) (msg-data msg))
           (SEND bcast self (cdr l))))))
```

Figure 1: Code for the mail queue which implements replication.

Dependability of a server will be increased by creating several exact copies of the server on different nodes. We refer to the copies as *replicas*. The task of supporting the replicas is divided between two actors: a distributor, which broadcasts messages to the replicas, and a collector, which takes the replicas' responses and extracts one response to send to the client. If either the distributor or the collector crashes, the replicas will be unable to communicate with the rest of the world. Therefore, the applicability of the given solution is limited due to the need for more dependable processors for running the distributor and collector. However, because these two

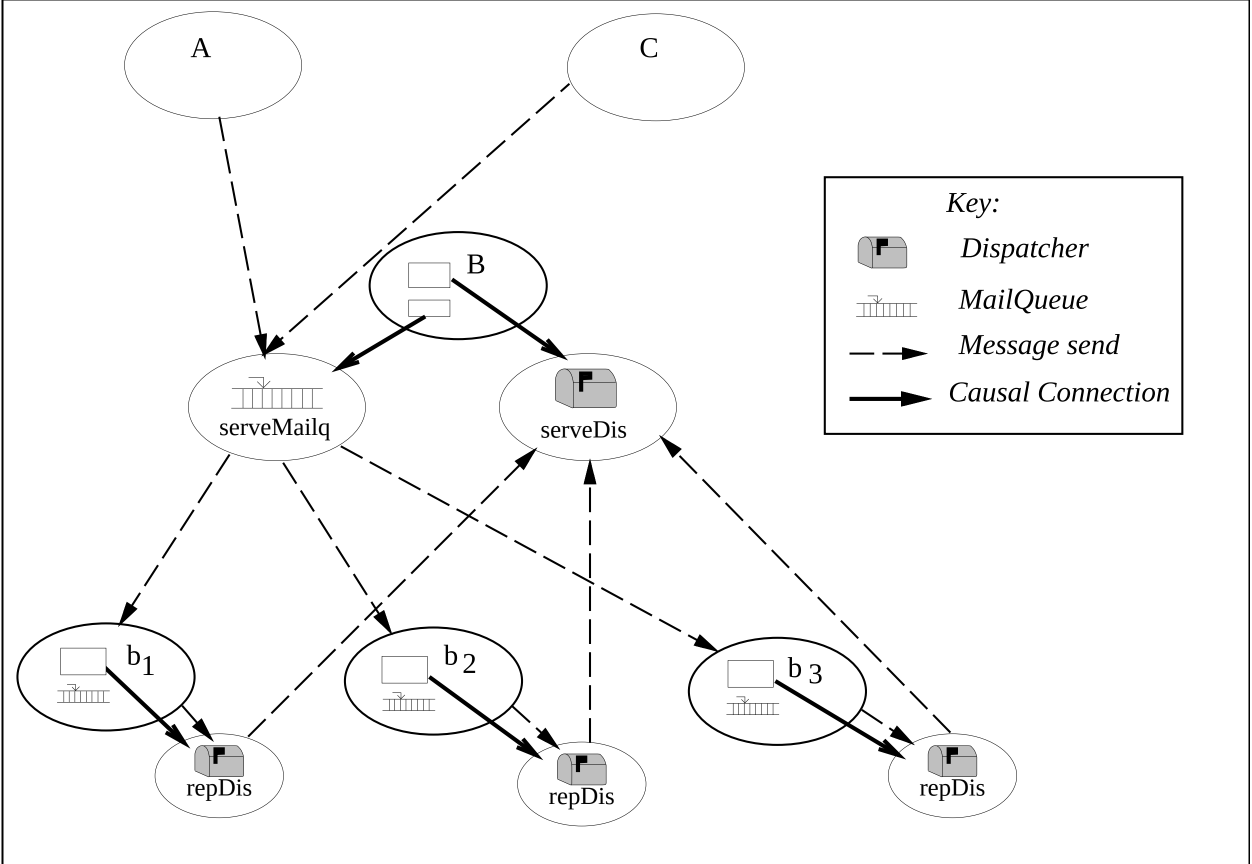

Figure 2: When a message is sent by the clients A or C to the replicated service B, the message is received by B's mail queue serveMailq (1). The message is then sent to each of the replicas (2).

objects are computationally much less expensive than the application itself, the solution allows us to increase the dependablilty of a system given a few highly-dependable processors. It is possible to use MAUD for implementing replication schemes which do not require a centralized distributor/collector; however we restrict our discussion to the centralized distributor/collector scheme.

We assume that managers themselves install appropriate meta-objects realizing a given dependability protocol. Therefore, we specify the relevant dependability protocols by describing the behavior of the initiating manager as well as the installed mail queues and dispatchers. A manager in charge of replicating a service takes the following actions to achieve the state shown in Figure 2:

1. The specified server is replicated by a manager by creating actors with the same behavior and acquaintance list (using HAL's `clone` function).

2. A mail queue is installed for the original server to make it act as the *distributor* described above. Messages destined for the original server are broadcast to the replicas. A broadcast using SSENDs is done so that all replicas receive messages in the same order and thus solve the same task.

3. The dispatcher of the original server is modified to act as the *collector* described above. The

first message out of each set of replica responses is selected to be passed to the destination. Since we assume only fail-stop failures, no complex voting scheme is necessary.

4. The dispatchers of the replicas are changed to forward all messages being sent to the original server's dispatcher. In addition, the messages are tagged so that the original server's dispatcher can eliminate multiple copies of the same message.

The dispatchers and mail queues are designed according to the specification in Section 4. The new mail queue for the original server is described in Figure 1. Note that message order is being preserved in the broadcast. We use HAL's `SSEND` function to guarantee consistent state at each replica. Figure 2 shows the resulting actions occurring when a message is sent to the replicated service. The original server is actor B. When a message is received by the distributor, $serveMailq$ (B's new mail queue), the message is broadcast to the replicas b_1, b_2, b_3. Each of the replicated actors has the same base-level behavior as B. Therefore, upon receipt of the message, each b_i responds in the same way B would have. However, if the replicas respond to the message, the message destinations would be rerouted by the dispatchers $repDis$ to the original server's dispatcher, $serveDis$ (serving as the collector). For each response, $serveDis$ gets three messages, one from each replica. It processes the three messages using some voting scheme and sends out a single response to the original destination. Note that the base-level actor B does not receive any messages now since all the incoming messages are redirected to the replicas by its mail queue $serveMailq$ and the outgoing messages are sent by the dispatchers of the replicas directly to its dispatcher $serveDis$.

Although this example is fairly simple, it does illustrate some of the benefits of our approach. The manager initiating the replication protocol needs no advance knowledge of the service to be replicated nor does the replicated service need to know that it is being replicated. Additionally, the clients using the replicated service are not modified in any way. These benefits give us the flexibility to dynamically replicate and unreplicate services while the system is running.

6 Composition of Dependability Characteristics

In some cases, dependability can only be guaranteed by using several different protocols. For example, a system employing replication to avoid possible processor faults may also need to guarantee consensus on multi-party transactions through the use of three-phase commit or some similar mechanism. Unfortunately, writing one protocol which has the functionality of multiple protocols can lead to very complex code. In addition, the number of possible permutations of protocols grows exponentially – making it necessary to predict all possibly needed combinations in a system. However, because the meta-components of an object are themselves objects in a reflective system, there is a general solution for composing two protocols using MAUD. A simple change to the meta-operations inherited from the `Object` class, along with a few restrictions on the construction of mail queues and dispatchers, allows us to layer protocols in a general way. Figure 3 shows how an *add-mailq* method could be expressed in terms of the other meta-operations to allow layering.

Because the current mail queue, `mailq`, and the current dispatcher, `dispatcher`, are objects, we can install meta-objects to customize their mail queue or dispatcher. By adding protocols in the above manner, the new mail queue functionality will be performed on incoming messages before they are passed on to the "old" mail queues. For the send behaviors, the process is reversed with the oldest send behavior being performed first and the newest behavior last, thereby creating an onion-like model with the newest layer closest to the outside world.

```
  (METHOD (add-mailq aMailq)
     (IF (null? mailq)
        (SEND change-mailq self aMailq)
        (SEND add-mailq mailq aMailQ)))
 (METHOD (add-dispatcher aDispatcher)
     (IF (null? dispatcher)
        (SEND change-dispatcher self aDispatcher)
        (SEND add-dispatcher dispatcher aDispatcher)))
```

Figure 3: The additional methods which must be inherited to allow for protocol composition.

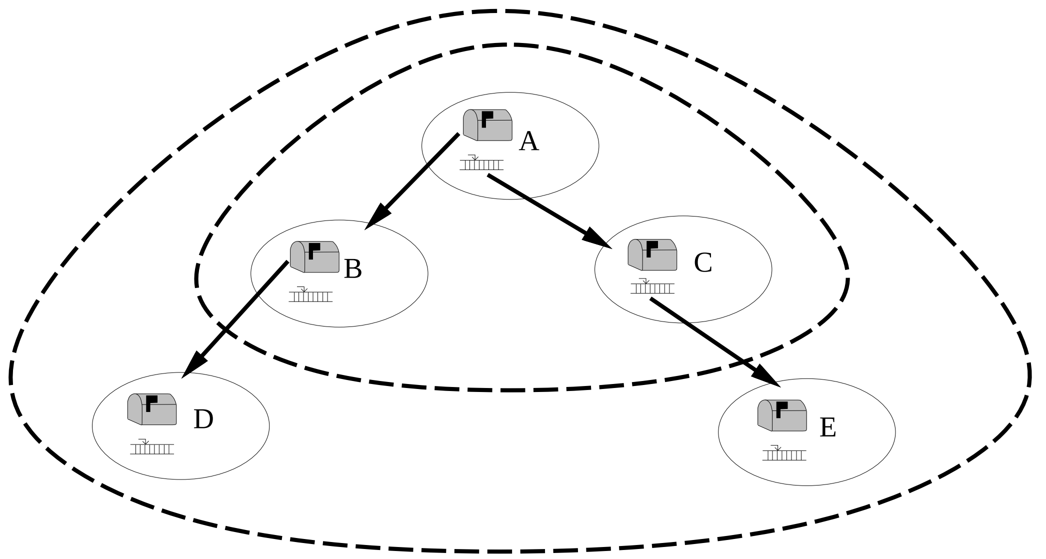

Figure 4: Partners and Owner relationships. A is the owner of all other actors in the figure. Dispatcher B and mail queue C are partners as well as dispatcher D and mail queue E.

To preserve the model, however, several restrictions must be applied to the behavior of dispatchers and mail queues. We define the *partner* of a mail queue as being the dispatcher which handles the output of a protocol and the partner of a dispatcher as being the mail queue which receives input for the protocol. In Figure 4, B and C are partners as well as E and D. Each pair implements *one* protocol. It is possible for a meta-object to have a null partner.

The *owner application* of a meta-object is inductively defined as either its base object, if its base object is not a meta-object, or the owner application of its base object. For example, in figure 4, A is the owner application of meta-objects B, C, D, and E. With the above definition we can restrict the communication behavior of the actors so that:

- A mail queue or dispatcher may send or receive messages from its partner or an object created by itself or its partner.
- Dispatchers may send messages to the outside world, i.e. to an object which is not a mail queue or dispatcher of the owner application (although the message might be sent through the dispatcher's dispatcher). Dispatchers may receive `transmit` messages from their base object.
- Mail queues may receive messages from the outside world (through its own mail queue) and send messages when responding to `get` messages from their base object.
- Objects created by a mail queue or dispatcher may communicate with each other, their creator, or their creator's partner.

Because of the above restrictions, regardless of the number of protocols added to an object there is exactly one path which incoming messages follow, starting with the newest mail queue, and exactly one path for outgoing messages in each object, ending with the newest dispatcher. Therefore, when a new dispatcher is added to an object, all outgoing messages from the object must pass through the new dispatcher. When a new mail queue is installed it will handle all incoming messages before passing them down to the next layer. Thus, a model of objects resembling the layers of an onion is created; each addition of a protocol adds a new layer in the same way regardless of how many layers currently exist. With the above rules, protocols can be composed without any previous knowledge that the composition was going to occur and protocols can now be added and removed as needed without regard not just to the actor itself, but also without regard to existing protocols. In figure 4, actors B and C are initially installed as one "layer". Messages come into the layer only through C and leave through B. Therefore, D and E may be installed with the `add-mailq` and `add-dispatcher` messages as if they were being added to a single actor. Now messages coming into the composite object through E and are then received by C. Messages sent are first processed by B and then by D.

7 Composition Example

In section 5, we demonstrated one method of implementing replication. We now build on that example to show how different protocols can be composed. Our system currently has three actors: A, B, and C, where B has been replicated and is currently represented by b_1, b_2, and b_3. The initial system configuration is shown in Figure 2.

Assume A initiates a transaction with databases B, and C. These transactions are implemented using a specific commit protocol. The commit protocol is chosen dynamically depending on the kinds of failures (site failures or communication failures [BHG87]) that need to be tolerated by the system. Assume that a two-phase commit protocol is implemented by a mail queue called *tpcMailq* and dispatcher called *tpcDis* installed at A, B and C. Also assume that A acts as the coordinator [BHG87] of the commit protocol. This scenario is depicted in Figure 5.

When A initiates a commit protocol, its *tpcDis* broadcasts a `vote-req` message to B and C, and waits for `vote` messages from them. The *tpcDis* of a participant sends a message to its

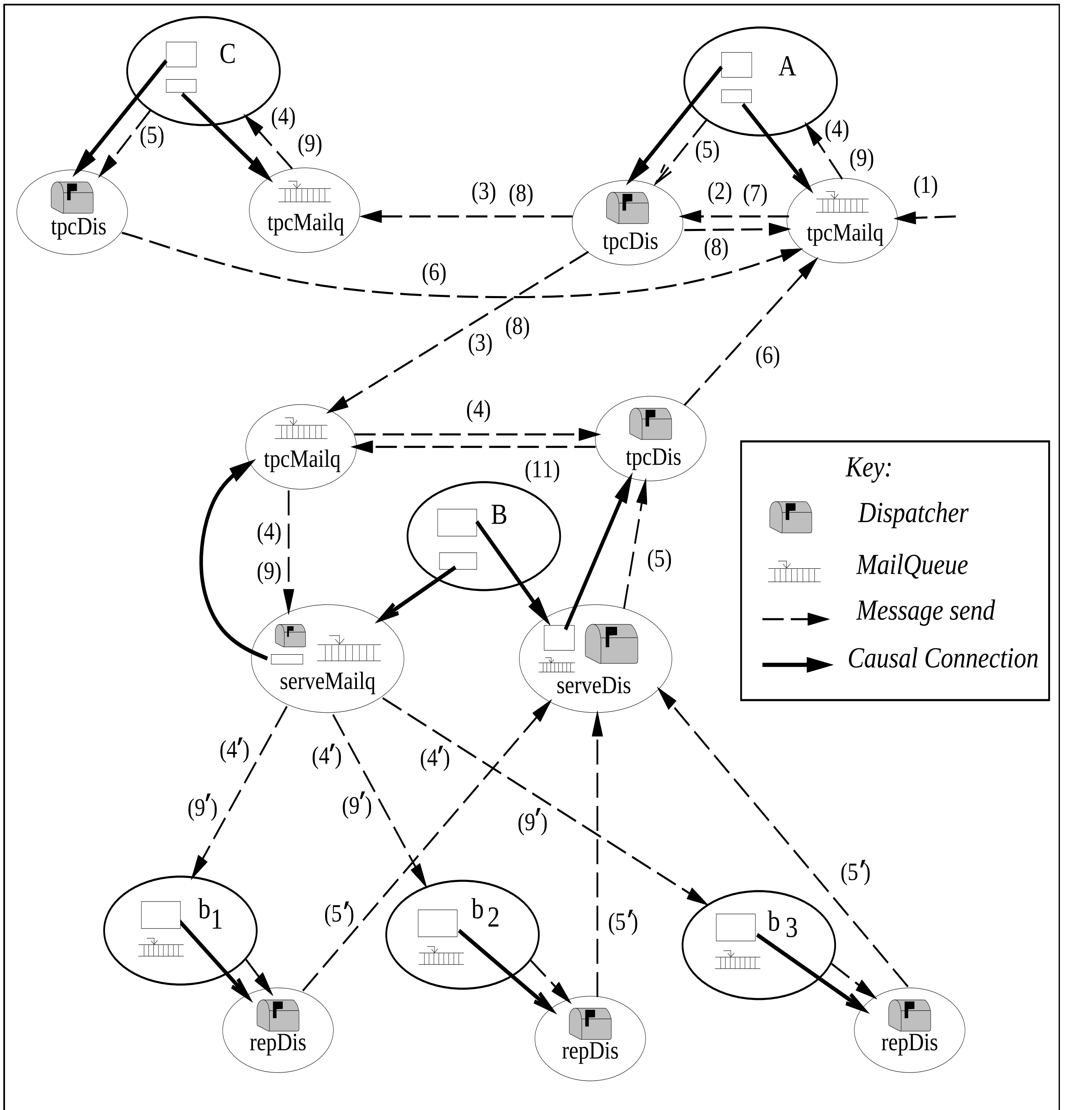

Figure 5: System resulting from the composition of two-phase commit with replication. When a transaction is received (1), *tpcMailq* notifies *tpcDis* (2). The *tpcDis* broadcasts a `vote-req` message (3). The vote is decided (4, 5) by the participants. The voting by the replicated actor B involves the messages $(4', 5')$. The decision of B and C is sent to the coordinator (6). The coordinator decides whether the transaction should be committed or aborted and sends its decision to the participants (7,8). If the final decision is to commit, the transaction is installed (9).

tpcMailq informing it of the voted value. If the participant voted `no`, its *tpcMailq* assumes that the transaction is aborted and allows subsequent messages to proceed. Otherwise, it waits for a `commit` or `abort` message from the coordinator. The votes of B and C are received by the *tpcMailq* of A and forwarded to its *tpcDis*. Based on the `vote` values, the *tpcDis* of A decides to commit or abort the transaction and broadcasts its decision. If the *tpcMailq* of B or C receives a message to commit the transaction, it sends a message to its base actor to install the transaction.

The applied meta-objects are designed to follow the restrictions given in Section 6 to preserve the onion-layer model. Figure 5 shows the resulting system after all three actors have the two-phase commit protocol installed. Because *tpcMailq* does not enqueue a message on data until after two-phase commit is resolved, a message sent to B after the commit protocol starts, is not copied and sent to the replicates until the protocol is finished. There are no differences between the way in which the two-phase commit protocol is installed on C and on the replicated B.

8 Discussion

The reflective capabilities described in this paper have been implemented in HAL [AW92]. Our work led to the addition of several categories of constructs to HAL. The structure for MAUD was built directly into HAL and the underlying system directs messages to dispatcher meta-objects, when sent, and to mail queue meta-objects when received. Additional functions to allow message manipulation were also added.

In our prototype implementation of MAUD, an application and its dependability protocols are linked together at compile time. It is currently not possible to load new protocols into an application at runtime. However, we are developing an execution environment containing a dynamic linker. Applications can invoke the dynamic linker and thereby be extended with new executable code: thus it is possible for an application to dynamically extend its repertoire of dependability protocols. Using dynamic linking instead of interpretation, dynamically constructed protocols have the same format and exhibit the same level of trustworthiness as statically linked protocols. In particular, it is possible to verify dynamically constructed protocols in the same way as statically linked protocols.

Several examples, including the ones given in this paper, have been implemented using HAL. Our experiments suggest that the performance cost of using the meta-objects is, in itself, not significant. The cost of the extra messages caused by meta-object to base-object communication is a constant factor; the meta-objects may be on the same node as their base-object, allowing these messages to be converted to function calls. The primary source of cost is that generalized protocols may be used instead of customized protocols which can exploit knowledge of a given application. An example of this cost is the inability of a general replication scheme to take advantage of commutable operations [MPS91]. Unfortunately, the cost due to generalizing protocols is difficult to express quantitatively since it depends on the application and the protocol: in some cases, the cost is quite high and, in others, insignificant. However, even if a programmer wants to exploit knowledge of an application, the customized protocols can be handled as any other protocol. Our methodology still preserves code modularity, dynamic protocol installation and removal, and composability with other protocols.

The dependability of applications developed using our methodology is highly dependent on the installation of "correct" meta-objects. As we have already explained, installation of erroneous meta-objects may have dramatic consequences for the behavior of an application. An important part of our future work is to come up with ways in which it is possible to reason about the behavior of meta-objects. A formal semantic framework is needed to verify the adherence of MAUD components to a given specification. Preliminary work in this area has been done in [VT93]. Safety would be addressed by having managers only install meta-objects that conform to some specification. The approach is also flexible and open-ended since many meta-objects may implement a given specification.

Besides protecting the meta-level, a manager may initiate the installation of meta-objects. Because we have the flexibility of dynamic protocol installation, self-monitoring could be constructed using daemons that, without human intervention, initiate protocols to prevent faulty behavior. The above capability will be especially useful in the future as systems grow and become proportionally harder for human managers to monitor. For example, daemons may use sensors to monitor the behavior of a system in order to predict potentially faulty components.

References

[ABB$^+$86] M. Acceta, R. Baron, W. Bolosky, D. Golub, R. Rashid, A. Tevanian, and M. Young. Mach: A New Kernel Foundation for UNIX Developement. In *USENIX 1986 Summer Conference Proceedings*, June 1986.

[Agh86] G. Agha. *Actors: A Model of Concurrent Computation in Distributed Systems.* MIT Press, 1986.

[Agh90] G. Agha. Concurrent Object-Oriented Programming. *Communications of the ACM*, 33(9):125–141, September 1990.

[AW92] G. Agha and K. Wooyoung. Compilation of a Highly Parallel Actor-Based Language. In U. Banerjee, D. Gelernter, A. Nicolau, and D. Padua, editors, *Proceedings of the Workshop on Languages and Compilers for Parallel Computing.* Yale University, Springer-Verlag, 1992. LNCS, to be published.

[BHG87] P. A. Bernstien, V. Hadzilacos, and N. Goodman. *Concurrency Control and Recovery in Database Systems.* Addison-Wesley, 1987.

[BJ87] K. P. Birman and T. A. Joseph. Communication Support for Reliable Distributed Computing. In *Fault-tolerant Distributed Computing.* Springer-Verlag, 1987.

[Coo90] E. Cooper. Programming Language Support for Multicast Communication in Distributed Systems. In *Tenth International Conference on Distributed Computer Systems*, 1990.

[DHW88] D. L. Detlefs, M. P. Herlihy, and J. M. Wing. Inheritance of Synchronization and Recovery Properties in Avalon/C++. *IEEE Computer*, 21(12):57–69, December 1988.

[FB88] Jacques Ferber and Jean-Pierre Briot. Design of a Concurrent Language for Distributed Artificial Intelligence. In *Proceedings of the International Conference on Fifth Generation Computer Systems*, volume 2, pages 755–762. Institute for New Generation Computer Technology, 1988.

[HA92] C. Houck and G. Agha. HAL: A High-level Actor Language and Its Distributed Implementation. In *Proceedings of th 21st International Conference on Parallel Processing (ICPP '92)*, volume II, pages 158–165, St. Charles, IL, August 1992.

[JM92] Sushil Jajodia and Catherine D. McCollum. Using Two-Phase Commit for Crash Recovery in Federated Multilevel Secure Database Management Systems. In *Proceedings of the 3rd IFIP Working Conference on Dependable Computing for Critical Applications*, pages 209–218, Mondello, Sicily, Italy, September 1992. IFIP. Preprint.

[LS82] Barbara Liskov and Robert Scheifler. Guardians and Actions: Linguistic Support for Robust, Distributed Programs. In *Conference Record of the Ninth Annual ACM Symposium on Principles of Programming Languages*, pages 7–19, Albuquerque, New Mexico, January 1982. ACM.

[Mae87] P. Maes. Computational Reflection. Technical Report 87-2, Artificial Intelligence Laboratory, Vrije University, 1987.

[MPS91] S. Mishra, L. L. Peterson, and R. D. Schlichting. Consul: A communication Substrate for Fault-Tolerant Distributed Programs. Technical report, University of Arizona, Tucson, 1991.

[OOW91] M. H. Olsen, E. Oskiewicz, and J. P. Warne. A Model for Interface Groups. In *Tenth Symposium on Reliable Distributed Systems*, Pisa, Italy, 1991.

[PS88] G. D. Parrington and S. K. Shrivastava. Implementing Concurrency Control in Reliable Distributed Object-Oriented Systems. In *Proceedings of the Second European Conference on Object-Oriented Programming, ECOOP88*. Springer-Verlag, 1988.

[Sch90] F. B. Schneider. The State Machine Approach: A Tutorial. *Lecture Notes in Computer Science*, 448:18–41, 1990.

[Smi82] B. C. Smith. Reflection and semantics in a procedural language. Technical Report 272, Massachusetts Institute of Technology. Laboratory for Computer Science, 1982.

[TS89] C. Tomlinson and V. Singh. Inheritance and Synchronization with Enabled-Sets. In *OOPSLA Proceedings*, 1989.

[VT93] Nalini Venkatasubramanian and Carolyn Talcott. A MetaArchitecture for Distributed Resource Management. In *Proceedings of the Hawaii International Conference on System Sciences*. IEEE Computer Society Press, January 1993. To Appear.

[WL88] C. T. Wilkes and R. J. LeBlanc. Distributed Locking: A Mechanism for Constructing Highly Available Objects. In *Seventh Symposium on Reliable Distributed Systems*, Ohio State University, Columbus, Ohio, 1988.

[WY90] T. Watanabe and A. Yonezawa. A Actor-Based Metalevel Arhitecture for Group-Wide Reflection. In J. W. deBakker, W. P. deRoever, and G. Rozenberg, editors, *Foundations of Object-Oriented Languages*, pages 405–425. Springer-Verlag, 1990. LNCS 489.

[YMFT91] Y. Yokote, A. Mitsuzawa, N. Fujinami, and M. Tokoro. The Muse Object Architecture: A New Operating System Structuring Concept. Technical Report SCSL-TR-91-002, Sony Computer Science Laboratory Inc., Feburary 1991.

[Yon90] A. Yonezawa, editor. *ABCL An Object-Oriented Concurrent System*, chapter Reflection in an Object-Oriented Concurrent Language, pages 45–70. MIT Press, Cambridge, Mass., 1990.