An Architecture for Multiagent Systems

An Approach Based on Commitments

Amit K. Chopra[1] and Munindar P. Singh[2]

[1] Università degli Studi di Trento akchopra.mail@gmail.com
[2] North Carolina State University singh@ncsu.edu

Abstract. Existing architectures for multiagent systems emphasize low-level messaging-related considerations. As a result, the programming abstractions they provide are also low level. In recent years, commitments have been applied to support flexible interactions among autonomous agents. We present a layered multiagent system architecture based on commitments. In this architecture, agents are the components, and the interconnections between the agents are specified in terms of commitments, thus abstracting away from low level details. A crucial layer in this architecture is a commitment-based middleware that plays a vital role in ensuring interoperation and provides commitment-related abstractions to the application programmer. Interoperation itself is defined in terms of commitment alignment. This paper details various aspects of this architecture, and shows how a programmer would write applications to such an architecture.

1 Introduction

An *architecture* is an abstract description of a system. The fundamental idea of an architecture is that it identifies *components* and their *interconnections* [1]. An *open* architecture is one that emphasizes the interconnections, leaving the components unspecified except to the extent of their interconnections. In this manner, an open architecture yields systems whose components can be readily substituted by other components.

When we understand multiagent systems from the standpoint of architecture, it is clear that the components are *agents* (or, rather, abstractly *roles*). Traditionally, the interconnections have been modeled in operational terms derived from an understanding of distributed systems. Consequently, the multiagent systems that result are over-specified and behave in an inflexible manner. With such systems, it is difficult to accommodate a richer variety of situations. What is required is the specification of interconnection in terms of higher-level abstractions.

For concreteness, we consider cross-organizational business processes as an application of multiagent systems that provide the happy mix of significance and complexity to demonstrate the payoff of using the proposed approach. The last few years have developed compelling accounts of the fundamental autonomy and heterogeneity of business partners, and the concomitant need to model these partners' interest. The related studies of interaction protocols hint at how we might engineer multiagent systems in such settings [2–4]. A feature of these approaches is their basis in commitments. Commitments yield a business-level notion of compliance: as long as agents discharge their commitments, they are free to interact as they please. However, the relationship of

protocols with architectures has not yet been adequately worked out. This requires an understanding of interoperability in terms of commitments.

We make a fresh start on multiagent systems via an architecture. Once we realize that we would only consider the components as agents understood flexibly, the associated interconnections must inevitably be the business relationships between the agents. One can imagine that some notional business value flows across such relationships, just as data flows over the traditional connectors of distributed computing. Thinking of the business relationships as interconnections yields an architecture for what we term *service engagements* [5].

The above architecture is conceptual in nature. Two natural questions arise: what programming abstractions does the architecture support, and how may we operationalize it over existing infrastructure that is no different from that underlying traditional approaches. Answering the above questions is the main contribution of this paper.

1.1 Middleware: Programming Abstractions

From a top-down perspective, an important layer of any architecture is middleware. Middleware supports programming abstractions for the architecture in a way that ensures interoperability between components in the architecture. A relatively simple middleware is one that provides reliable message queuing services, freeing the programmer from the burden of, for example, implementing persistent storage and checking for acknowledgments. These days, reliable message queuing is just one of many abstractions supported in enterprise middleware. In cross-organizational business processes, the common middleware is centered on the abstractions of messaging. The resulting architectural style is termed the *Enterprise Service Bus (ESB)*. ESBs emphasize messaging abstractions and patterns—for example, Apache Camel supports the enterprise integration patterns in [6]. Further, ESBs support an event-driven architecture so as to promote loose coupling between business applications. ESBs provide various kinds of translation services, routing, and security, among other things, thus saving the application programmer a good deal of repetitive effort. Some ESB implementations, such as provided by Oracle, also support business protocols such as RosettaNet [7].

Ideally, middleware should offer abstractions that follow closely the vocabulary of the domain. ESBs purport to support business applications; however, they lack business-level abstractions. The abstractions they support, e.g., for RosettaNet, involve message occurrence and ordering but without regard to the meanings of the messages. Thus RosettaNet can be thought of as a protocol grammar. Other protocols, e.g., Global Data Synchronization Network (GDSN) [8], would correspond to alternative grammars. Each grammar is arbitrary and its correctness or otherwise is not up for consideration.

Figure 1 shows the conceptual arrangement of a service-oriented architecture based on such ESBs. Programmers design business *processes* (for example, in BPEL) based on a public interface specification (for example, in WS-CDL or based on a protocol such as RosettaNet). Messaging-based middleware, such as described above, hides the details of the infrastructure from process programmers.

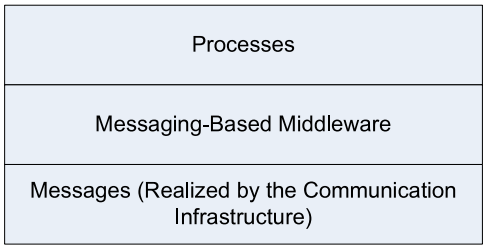

Fig. 1. Current enterprise middleware, conceptually

1.2 Overview of Approach

We assume a conventional infrastructure based on messaging, such as is already made available by middleware such as the Java Messaging Service and specified in the emerging standard known as the Advanced Message Queuing Protocol (AMQP) [9]. This infrastructure supports point-to-point messaging over channels that preserve pairwise message order and guarantee eventual delivery. It is important to emphasize that such infrastructure is commonly available in existing implementations.

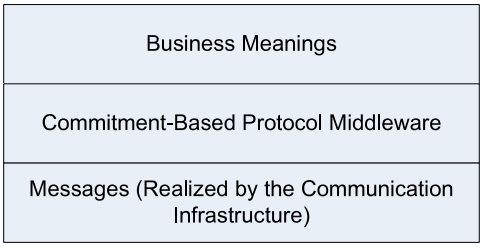

Fig. 2. Commitment middleware, conceptually

The essential idea underlying our approach is that we can thus view system architecture at two levels of abstraction: business and infrastructure. The business level deals with meaning whereas the infrastructure provides the operationalization. Accordingly, we view the function of middleware as bridging this conceptual gap. Figure 2 shows that our middleware lies in the middle between meaning and messaging. In our approach, business meaning is expressed in terms of commitments. Commitments arise in virtually all cross-organizational business applications. Thus, reasoning about commitments would be applicable to all of them. Commitments underlie two correctness criteria: *compliance* and *alignment*. Agents are compliant as long as they discharge their commitments; such a notion of compliance naturally takes into account agents' autonomy. Agents are aligned as long they agree on whatever commitments as may result from their communications. Alignment is, in fact, a key form of business interoperability [10, 11].

The proposed middleware provides commitment-based abstractions. The middleware supports not only the basic commitment operations [12], but also high-level patterns that build on the commitment operations. The middleware ensures that if applications are determined interoperable at the level of business meaning, then infrastructure-level concerns such as asynchrony do not break the interoperability.

1.3 Contributions

The contribution of this paper lies in making explicit the architecture that commitment alignment, as a notion of business-level interoperability, supports. This includes the specification of agent interfaces and the question of their compatibility, the design of the middleware that bridges between the business level and the infrastructural level, and the programming abstractions that are made available to application programmers. In all of these elements, the architecture presented is significantly different from current multiagent architectures. We also outline how the middleware may be extended with common business patterns and additional kinds of alignment. To illustrate how the architecture supports a new way of programming, consider that traditional agent communication is treated simply in terms of sending and receiving messages. However, with our proposed architecture, it would be possible to encode communication in terms of commitment operations and patterns. The benefit accrued is of a nature similar to that accrued by being able to write agents (their internal reasoning, in particular) in terms of BDI abstractions rather than low-level procedural abstractions.

The rest of this paper is organized as follows. Section 2 describes commitments formally, what alignment means, and why misalignments occur. Some misalignments can be detected at the level of business meanings by a static analysis of the interfaces, whereas others that occur due to the nature of distributed systems must be prevented by careful design of the middleware. Section 3 describes an architecture based on commitments. It describes the components and the interconnections and the layers in the architecture. Section 4 describes a sample set of useful patterns that the middleware supports. Section 5 discusses the relevant literature.

2 Commitment Alignment

Interoperability among participants means that each participant fulfills the expectations made by the others. To understand an architecture, it is important to understand what interoperability in the architecture means. In our approach, an agent represents each participant, and the expectations of an agent take the form of commitments. Existing work on service interoperability treats expectations solely at the level of messages [13–15].

Let us explain how commitments yield expectations. A commitment is of the form $\mathsf{C}(debtor, creditor, antecedent, consequent)$, where $debtor$ and $creditor$ are agents, and $antecedent$ and $consequent$ are propositions. This means that the debtor commits (to the creditor) to bringing about the consequent if the antecedent holds. For example, $\mathsf{C}(EBook, Alice, \$12, BNW)$ means that EBook commits to Alice that if she pays \$12, then EBook will send her the book *Brave New World*. Agents interact by sending each

other messages. The messages have meanings in terms of how they affect the agents' commitments toward each other. For example, an offer message from EBook to Alice may bring about the aforementioned commitment.

Now imagine that at some point in their interaction, Alice infers that EBook is committed to sending her the book she paid for, but EBook infers no such commitment. Their interaction would break down at the level of business meaning. In other words, Alice and EBook would not be interoperable. In general, a key requirement for interoperability is that the interacting agents remain aligned with respect to their commitments. Commitment alignment is a key form of business-level interoperability. Agents are aligned if whenever one agent (as creditor) infers a commitment from a second agent, the second agent (as debtor) also infers that commitment. If we can guarantee *a priori* that agents never—at no point during any possible interaction—get misaligned, only then the agents are interoperable.

In general, agents may get misaligned because of their *heterogeneity*, *autonomy*, and *distribution*.

Heterogeneity Agents may assign incompatible meanings to the messages they are exchanging. To be able to successfully interact, the agents must agree on what their communications count as. Heterogeneity is the cause of misalignment in Example 1.

Example 1. For Alice, an *Offer* message from EBook counts as a commitment from EBook to ship a book in return for payment. Whereas for EBook, *Offer* does not count as any such commitment; but an explicit *Accept* from Alice does. Thus, when EBook sends Alice an *Offer* message, Alice infers the commitment, but EBook does not—a misalignment.∎

Heterogeneity is addressed by statically analyzing if the interfaces of agents are compatible [10].

Autonomy Agent autonomy must be accommodated; however, accommodating autonomy is nontrivial. The reason is that autonomy operationally means that they are free to send messages. In turn, this means that communication between agents is asynchronous. Thus, in general, agents may observe messages in different orders. Since messages are understood in terms of their effects on commitments, the agents involved may become misaligned. This is the cause of misalignment in Example 2.

Example 2. EBook sends an *Offer* to Alice, where the offer means a commitment that if Alice pays, then EBook will send the book. Alice sends the payment (message) for the book. Concurrently, EBook cancels the offer by sending *CancelOffer*. Alice observes EBook's cancellation after sending the payment; so she regards it as spurious. EBook observes Alice's payment after sending its cancellation, so EBook considers the payment late. As a result, Alice infers that EBook is committed to sending her the book, but EBook does not infer that commitment. Thus, EBook and Alice are misaligned. ∎

An ideal approach to addressing the challenge of autonomy should work without curbing autonomy. In contrast, existing approaches to reasoning about commitments in distributed systems typically rely on some kind of synchronization protocol; synchronization, however, inhibits autonomy. Chopra and Singh [11] formalize the inferences made upon observing commitment-related messages in such a way that, in spite of autonomy, agents remain aligned.

Distribution In a distributed system, some agents may have more information about relevant events than others. This is the cause of misalignment in Example 3.

Example 3. Alice commits to Bob that if the sky is clear at 5PM, then she will meet him at the lake. At 5PM, Bob observes (a message from the environment) that the sky is clear, and therefore infers that Alice is unconditionally committed to meeting him at the lake. However, Alice does not know that the sky is clear, and therefore does not infer the unconditional commitment. Bob and Alice are thus misaligned. ∎

Chopra and Singh [11] state *integrity constraints*, which are constraints upon agent behavior necessary to handle distribution. The constraints are of two kinds: (1) a debtor must inform the creditor about the discharge of a commitment, and (2) a creditor must inform the debtor about the detach of a commitment. One should not consider alignment until such information has been propagated.

2.1 Characterizing Alignment

A set of agents is aligned if in all executions, at *appropriate* points during their execution, if a creditor infers a commitment from its observations, the debtor also infers the commitment from its own observations [11]. An "appropriate" point in the execution of a multiagent system is given by consistent observations of the various agents where two additional properties hold. One, alignment may only be considered at those points where no message is in transit. Such points are termed *quiescent*. Two, alignment may only be considered at those points that are *integral* with respect to the stated information propagation constraints. The motivation behind the above properties is simply that it would surprise no one if two agents failed to infer matching commitments when they had made differing observations: either because some message was in transit that only its sender knew about or because some message was not sent, and some agent had withheld material facts from another.

2.2 Background on Commitments

A commitment is of the form $\mathsf{C}(x, y, r, u)$ where x and y are agents, and r and u are propositions. If r holds, then $\mathsf{C}(x, y, r, u)$ is *detached*, and the commitment $\mathsf{C}(x, y, \top, u)$ holds. If u holds, then the commitment is *discharged* and doesn't hold any longer. All commitments are *conditional*; an unconditional commitment is merely a special case where the antecedent equals $\top$. Singh [16] presents key reasoning postulates for commitments.

The commitment operations are reproduced below (from [12]). CREATE, CANCEL, and RELEASE are two-party operations, whereas DELEGATE and ASSIGN are three-party operations.

– CREATE(x, y, r, u) is performed by x, and it causes $\mathsf{C}(x, y, r, u)$ to hold.
– CANCEL(x, y, r, u) is performed by x, and it causes $\mathsf{C}(x, y, r, u)$ to not hold.
– RELEASE(x, y, r, u) is performed by y, and it causes $\mathsf{C}(x, y, r, u)$ to not hold.
– DELEGATE(x, y, z, r, u) is performed by x, and it causes $\mathsf{C}(z, y, r, u)$ to hold.
– ASSIGN(x, y, z, r, u) is performed by y, and it causes $\mathsf{C}(x, z, r, u)$ to hold.

Let us define the set of messages that correspond to the basic commitment operations. Let Φ be a set of atomic propositions. In the commitment operations, r and u are formulas over Φ using $\wedge$ and $\vee$. $Create(x, y, r, u)$ and $Cancel(x, y, r, u)$ are messages from x to y; $Release(x, y, r, u)$ from y to x; $Delegate(x, y, z, r, u)$ from x to z; and $Assign(x, y, z, r, u)$ from y to x. Suppose $c = \mathsf{C}(x, y, r, u)$. Then $Create(c)$ stands for $Create(x, y, r, u)$. We similarly define $Delegate(c, z)$, $Assign(c, z)$, $Release(c)$, and $Cancel(c)$. $Inform(x, y, p)$ is a message from x to y, where p is conjunction over Φ. Observing an $Inform(p)$ causes p to hold, which may lead to the discharge or detach of a commitment.

Below, let $c_B = \mathsf{C}(EBook, Alice, \$12, BNW)$; $c_G = \mathsf{C}(EBook, Alice, \$12, GoW)$; $c_0 = \mathsf{C}(EBook, Alice, \$12, BNW \wedge GoW)$. ($BNW$ stands for the book ***Brave New World***; GoW stands for the book ***Grapes of Wrath***).

3 Multiagent System Architecture

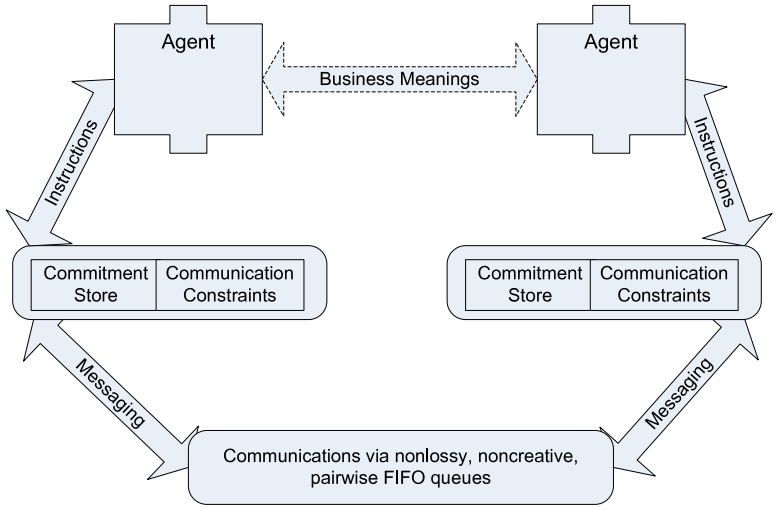

Fig. 3. Understanding Commitment-Based Architecture

Figure 3 shows our proposal for a multiagent system architecture. At the level of business meaning, the components are the agents in the system representing the interacting business partners. As pertains to programming using our architecture, at the

top, we have agents and at the bottom the communication layer; the middleware sits in between. This layered architecture is characterized by three kinds of interfaces.

- At the business level, the interface is between agents and is expressed via meanings. The business analyst and the software developer who programs using commitments would think at this level (of business relationships), and would be unaware of any lower layer.
- At the implementation level, the interface is between our middleware and the communication infrastructure and is based on traditional messaging services. In a traditional distributed system, a software developer would need to think at this level. In our approach, only the implementor of our middleware thinks at this level.
- Between the agent and the middleware, the interface is largely in terms of instructions from the agent to the middleware: when an agent tells the middleware to apply a commitment operation or one of the additional patterns based on commitments such as *Escalate* (patterns described later).

Two nice features of our approach are that (1) the instructions use the same vocabulary as the business meanings and (2) we specify middleware that guarantees alignment as long as the instructions are limited to the commitment operations or patterns. Below, we describe each components (agents), interconnections (interfaces), and layers in detail.

3.1 Agents

Agents represent business partners. They provide and consume real-world services by participating in service engagements. The principal elements of interest in an agent are its interface and its reasoning engine.

Interface An agent's interface describes the messages it expects to exchange with other agents, along with the business meanings of such messages. Table 1 shows Alice's interface specification. The left column is the actual protocol whereas the right column shows a traditional understanding of those messages. For example, $Create(EBook, Alice, \$12, BNW)$ message from EBook to Alice corresponds to an offer from EBook.

Table 1. An example interface for Alice

Commitment Protocol Message	**Traditional Message**
$Create(EBook, Alice, \$12, BNW)$	*Offer*$(EBook, Alice, \$12, BNW)$
$Create(Alice, EBook, BNW, \$12)$	*Accept*$(Alice, EBook, BNW, \$12)$
$Release(EBook, Alice, \$12, BNW)$	*Reject*$(Alice, EBook, \$12, BNW)$
$Inform(EBook, Alice, BNW)$	*Deliver*$(EBook, Alice, BNW)$
$Inform(Alice, EBook, \$12)$	*Pay*$(Alice, EBook, \$12)$

The interface shown in Table 1 is admittedly quite simple in that it does not talk about the penalties for the violation or cancellation of a commitment. Penalties would be encoded as additional commitments in the interface.

Notice that the interface does not contain some of the procedural constructs commonly found in interface description languages or protocols, such as sequence, choice, and so on. For example, it does not say, that upon observing an offer Alice has a *choice* between accepting or rejecting the offer—there is simply no need to say so. A rejection sent after Alice accepts the offer and EBook sends the book should have no effect—Alice should remain committed to pay. The formalization of commitments in [11] captures such intuitions, and makes the statement of procedural constructs in interfaces largely unnecessary. A second reason such constructs are introduced is to simply make the interaction synchronous. However, such constructs are rendered superfluous by the approach for reasoning about commitments in asynchronous settings [11]. Finally, threading constructions such as fork and join are clearly implementation details, and have no place in an interface. Of course, if an application demands a procedural construct, it could be introduced. For example, Alice may not trust booksellers and her interface might constrain delivery of books before payment. Alice will then be noninteroperable at the messaging-level with booksellers who require payment first; however, it would not affect alignment, that is, commitment-level interoperability [17].

As described earlier, misalignments arise when agents ascribe incompatible meanings to messages. An application programmer would specify an interface and publish it. Before interacting with other agents, the agent would presumably check for compatibility with the other agents.

Engine The engine drives the agent. It represents the private policies of the agent; these govern when an agent should pass an instruction to the middleware, how instruction parameters should be bound, how an agent should handle returned callbacks (described below) and so on. In fact, the engine is the place for all the procedural details. For example, Alice's policy may enforce a choice between accept and reject upon receiving an offer, or dictate that payment be sent only after receiving books.

Writing the engine is where the principal efforts of a programmer are spent. The implementation of the engine could take many forms. It could be a BPEL, Jess, JADE, or a BDI implementation such as Jason, for example. The details are irrelevant as long as it is consistent with reasoning about commitments.

From the programming perspective, the engine is coded in terms of the *meaning* of a message, not the message itself. In other words, the API that the programmer uses to interface with the middleware is in terms of commitment operations and other patterns built on top of the commitment operations. When the meaning concerns the sending of the message, the meaning may be thought of as an instruction (API) from the agent's engine to the middleware. The middleware then sends the appropriate messages. Analogously, for an incoming message, the engine registers a callback with the middleware that returns when the commitment operation corresponding to the message has been executed. Thus, the programmer's API is a business-level one, one of the goals we set out to achieve.

3.2 Middleware

To relate meanings to messages, the middleware takes on the responsibility for representing and reasoning about commitments. The middleware consists of a commitment

reasoner, maintains a commitment store, and is configured with communication constraints needed for the commitment operations and the further patterns (described later). The middleware computes commitments as prescribed in [11], and thus ensures that no misalignments arise because of autonomy and distribution. Further, as described above, the middleware's interface with the agent is instruction and callback-based.

The commitment reasoner presents a query interface to the agent (specifically the agent's engine), which can be used to inquire about commitments in the store. The engine can use such a information to decide on a course of action. For example, Alice's policy might be such that she sends payment only if $\mathsf{C}(EBook, Alice, \$12, BNW)$ holds.

The middleware maintains a *serial*, point-to-point communication interface with each other agent in the system through the communication layer. This means that an agent's middleware processes messages involving another particular agent—sent or received—one at a time. This is necessary to ensure consistency of the commitment store.

3.3 Communication Layer

The role of the communication layer is to provide reliable, ordered, and noncreative delivery of messages. Reliability implies that each sent message is eventually delivered; ordered implies that any two messages sent by an agent to another will arrive in the order in which they were sent, and noncreative means messages are not created by the infrastructure. Such a communication layer can be readily implemented by available reliable message queuing solutions.

3.4 Programming the Middleware: Example Scenario

Going back to our purchase example, suppose EBook wishes to sell BNW to Alice. The scenario may be enacted as follows.

1. EBook computes that it wants to make Alice an offer for BNW on internal grounds, such as excess inventory or the goal of making a profit.
2. At the level of business meaning, EBook sends an offer to Alice. At the computational level, this is effected by EBook instructing its middleware to create the corresponding commitment $\mathsf{C}(EBook, Alice, \$12, BNW)$.
3. The middleware then sends Alice corresponding message $Create(EBook, Alice, \$12, BNW)$.
4. The message travels along the communication infrastructure and arrives at Alice's endpoint of the message queue.
5. At the computational level, Alice's middleware receives the message from the communication layer, computes the corresponding commitment, and triggers Alice's callback on the creation of that commitment to return, in effect returning to the business level.
6. Alice may reason on the commitment and may decide to accept the offer, based on her private considerations such as goals.

Table 2. A snippet of EBook's code

```
if (preferredShopper(shopper) and inStock(book)) {

    Proposition price = lookupPrice(book);

    //register handler with middleware for accept from the shopper
    register(created(shopper, EBook, book, price), handler1);

    //Send an offer to the shopper
    Create(EBook,shopper,price,book);

   }
...

//Handler for accept
handler1(Debtor shopper, Proposition book, Proposition price) {

   //shopper accepted, so send the book
   Inform(EBook, shopper, book);
}
```

7. Alice responds by accepting—by instructing her middleware to create $\mathsf{C}(Alice, EBook, BNW, \$12)$; and so on.

Table 2 shows sample code for EBook. Although it is not possible to write such code yet (as the middleware hasn't been implemented yet), it captures the spirit of programming with a business-level API. In other examples, the more complex communication constraints would also apply.

4 Abstractions Supported by the Middleware

As mentioned before, the middleware supports sending notifications to debtors and creditors about detaches and discharges, respectively. The middleware also supports other integrity constraints critical to alignment. The middleware supports all commitment operations, including delegation and assignment, which are three-party operations, and guarantees that even in asynchronous settings, the operations occur without giving causing misalignments. Alignment is guaranteed because the middleware embodies the techniques for alignment developed in [11]. Here, we discuss even high-level abstractions in the form of commitment patterns and additional forms of alignment that the middleware could practically support.

4.1 Patterns

We sketch some of the patterns here; these derive from those presented by Singh *et al.* [5]. Below, we describe a sample set of patterns that can readily be supported by the middleware.

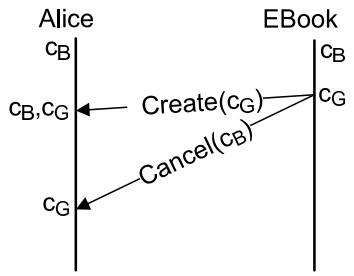

Fig. 4. Update pattern

Figure 4 shows the pattern for updating a commitment. At the programming level, this corresponds to the debtor sending an *Update* instruction to the middleware. At the computational level, the debtor's middleware sends two messages: one to cancel the existing commitment, and another to create a new commitment in its place.

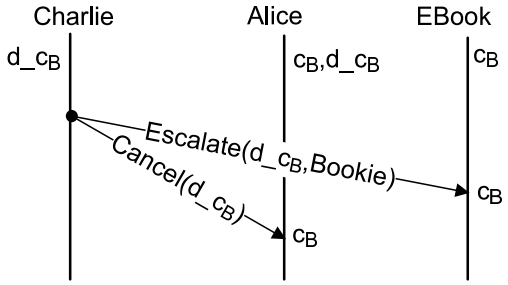

Fig. 5. Escalate pattern

Figure 5 shows the pattern for escalating a delegated commitment. Consider that Alice has delegated c_B to Charlie. (In the figures, a commitment with the name prefix $d\_$ is the delegated version of a commitment. Since $c_B = \mathsf{C}(EBook, Alice, \$12, BNW)$, in Figure 5, $d\_c_B = \mathsf{C}(Charlie, Alice, \$12, BNW)$.) The delegatee (Charlie) may find itself unable to fulfill the commitment. Here, the delegatee sends an *Escalate* instruction to the middleware. The middleware then sends a message notifying the delegator of the escalation of the commitment, and a $Cancel$ message to the creditor.

Figure 6 shows the pattern for delegating a commitment without retaining responsibility. Here, the debtor instructs the middleware to accomplish *DelegationWithoutRe-*

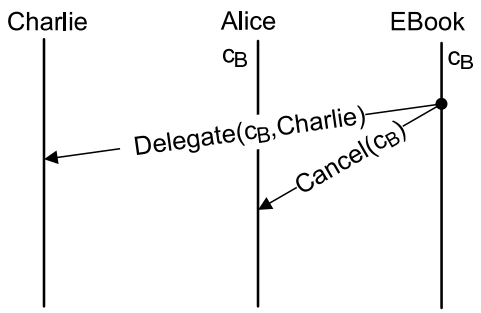

Fig. 6. Delegating without responsibility pattern

sponsibility. Along with the $Delegate$ instruction to the delegatee, the middleware sends a $Cancel$ message to the creditor thus absolving the debtor of any further responsibility. (Presumably, upon receiving the $Delegate$, Charlie will send $Create(d\_c_B)$ to Alice.)

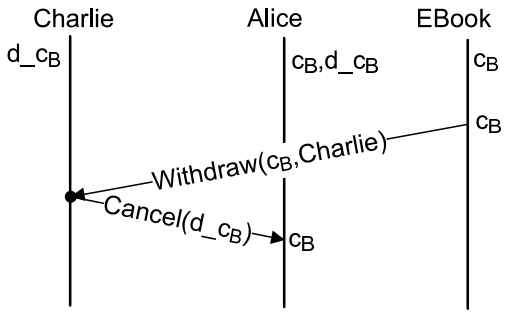

Fig. 7. Withdraw pattern

Figure 7 shows the pattern for withdrawing a delegated commitment. The delegator sends a *Withdraw* instruction to the middleware. The middleware then sends a $Withdraw$ message to the delegatee. The delegatee's middleware, upon receiving this message, sends a $Cancel$ to the creditor. The callback for *Withdraw* would return in the delegatee.

Figure 8 shows the pattern for division of labor: different parts of the commitment are delegated to different parties. Here, the delivery of BNW is delegated to Charlie and that of GoW is delegated to Barnie.

4.2 Other Forms of Alignment

Alignment as described in the previous sections and in [11] is essentially a creditor-debtor relation. When a creditor-debtor misalignment arises, there is the possibility of a violation of a commitment, and therefore, noncompliance. The following additional

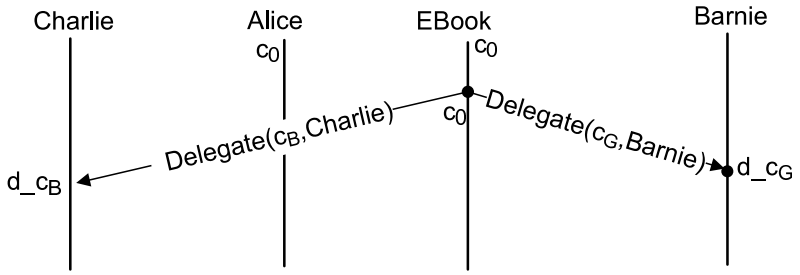

Fig. 8. Division of labor pattern

forms of alignment may be supported as additional patterns in the middleware. These forms of alignment may not necessarily result in noncompliance as it relates to commitment violation; nonetheless, these forms are useful for maintaining coherence in virtual organization settings, and are commonly effected in practice.

Debtor-debtor Alignment For example, suppose EBook delegates the commitment to send Alice a book to another bookseller Charlie. Then, EBook might want to be notified when Charlie discharges the commitment by sending the book, and vice versa.

Such alignment may be formalized in terms of debtor-debtor alignment: two agents who are related by a delegation relation remain aligned with respect to the discharge of the commitment. To effect such alignment would mean that the middleware would have to be configured with the additional constraint that if a debtor delegates a commitment to another agent, then whenever one of them discharges the commitment, it notifies the other.

Considering alignment in a *debtor group* could also be a useful notion. When two or more agents are committed for the same thing (thus the group), then whenever one discharges the commitment, it notifies the entire group.

Creditor-creditor Alignment In a similar vein, suppose Alice assigns the commitment made to her by EBook to Bob. Alice may want to be notified when Bob sends the payment, and vice versa.

This alignment is between creditors, and it is formalized and effected analogously to debtor-debtor alignment.

Contextual Alignment Each commitment has a social or legal context. Although we have omitted the context from the commitment so far, each commitment is in general a relation between three agents, the debtor, the creditor, and the context, and is expressed as $\mathsf{C}(debtor, creditor, context, antecedent, consequent)$. The context's role is the enforcement of the commitment. If EBook and Alice are operating on eBay, then eBay is the context of their interaction. Applications such as eBay, in which the context itself plays an active role, typically have the requirement that the context should also be aligned with respect to the commitment.

Contextual alignment involves three parties; stronger guarantees, such as causal delivery [18] may be required from the communication layer.

5 Discussion: Conclusions and Future Work

In this paper, we have presented a multiagent system architecture based on interaction and commitments. CSOA as an architectural style was first proposed in [5]; the current paper elaborates on that theme by taking into account the results on commitment alignment [11]. In particular, we have discussed a middleware that can compute commitments and guarantee alignment between agents even in completely asynchronous settings. Notably, the middleware provides high-level programming abstractions that build on commitment operations. We have also sketched alternative kinds of alignments that the middleware could practically support, thus further alleviating the programmer's burden.

Our architecture is unique in that commitments form the principal interconnections between agents. We deemphasize the implementation of the agent's engine. Agent programming languages, for example 2APL [19], remain largely based on BDI and do not support commitments. As mentioned before, such languages can be used to create an agent's engine. Enhancing agent programming frameworks such as JADE with high-level abstractions, for example, as illustrated in [20], is no doubt useful. However, when one talks of multiagent systems, that invariably involves interaction and commitments. Therefore, a programming language or a framework for multiagent systems should ideally support reasoning about commitments, and have commitment-related abstractions. Even when interactions protocols are supported in agent-oriented methodologies and platforms, it is at the level of specific choreographies, and not of meaning (for example, [21–24]). Tropos [25] uses the notion of goals to abstract away from agent reasoning and specific plans; however, when it comes to architectural specifications, Tropos resorts to data and control dependencies. Dastani *et al.* [26] show how to model a rich family of coordination connectors for multiagent systems, formalized as data and control flow abstractions. They do not consider the meaning of messages and thus lack the business-level semantics that distinguishes our work. Winikoff supports commitments in SAAPL by providing mappings from commitments to BDI-style plans, but the commitment reasoning supported is fairly limited [4]. Fornara *et al.* [27] base the semantics of communicative acts in terms of commitments. However, the operational semantics of commitments themselves do not consider asynchronous settings. As a consequence, an architecture based on the communicative acts would require strong assumptions such as synchrony.

In general, the treatment of communication at a low level in terms of data and control flow is complementary to our work. We can explain this in terms of correctness properties [17]. At a business level, commitment alignment is the correctness property as pertains to interoperability; at the level of data and control, interoperability is often characterized in terms of liveness and safety. On the one hand, agents may be aligned (with respect to their commitments) but deadlocked, each waiting for the other to take the next step. On the other hand, agents may be deadlock-free but misaligned.

The main priority for our research is the implementation of the proposed architecture. The language used here to give meanings to communications is sufficiently expressive for our purposes. We are investigating more powerful languages, however, for more subtle situations.

Acknowledgments. This research was partially supported by the Integrated FP6-EU project SERENITY contract 27587.

References

1. Shaw, M., Garlan, D.: Software Architecture: Perspectives on an Emerging Discipline. Prentice-Hall, Upper Saddle River, NJ (1996)
2. Yolum, P., Singh, M.P.: Flexible protocol specification and execution: Applying event calculus planning using commitments. In: Proceedings of the 1st International Joint Conference on Autonomous Agents and MultiAgent Systems, ACM Press (July 2002) 527–534
3. Desai, N., Mallya, A.U., Chopra, A.K., Singh, M.P.: Interaction protocols as design abstractions for business processes. IEEE Transactions on Software Engineering **31**(12) (December 2005) 1015–1027
4. Winikoff, M.: Implementing commitment-based interactions. In: Proceedings of the 6th International Joint Conference on Autonomous Agents and Multiagent Systems. (2007) 1–8
5. Singh, M.P., Chopra, A.K., Desai, N.: Commitment-based SOA. IEEE Computer **42** (2009) Accepted; available from http://www.csc.ncsu.edu/faculty/mpsingh/papers/.
6. Hohpe, G., Woolf, B.: Enterprise Integration Patterns: Designing, Building, and Deploying Messaging Solutions. Addison-Wesley Longman Publishing Co., Inc. (2003)
7. RosettaNet: Home page (1998) www.rosettanet.org.
8. http://www.gs1.org/productssolutions/gdsn/: GDSN
9. AMQP: Advanced message queuing protocol (2007) http://www.amqp.org.
10. Chopra, A.K., Singh, M.P.: Constitutive interoperability. In: Proceedings of the 7th International Conference on Autonomous Agents and Multiagent Systems. (2008) 797–804
11. Chopra, A.K., Singh, M.P.: Multiagent commitment alignment. In: Proceedings of the 8th International Joint Conference on Autonomous Agents and MultiAgent Systems (AAMAS), Columbia, SC, IFAAMAS (May 2009) 937–944
12. Singh, M.P.: An ontology for commitments in multiagent systems: Toward a unification of normative concepts. Artificial Intelligence and Law **7** (1999) 97–113
13. Fournet, C., Hoare, C.A.R., Rajamani, S.K., Rehof, J.: Stuck-free conformance. In: Proceedings of the 16th International Conference on Computer Aided Verification (CAV). Volume 3114 of LNCS., Springer (2004) 242–254
14. Bravetti, M., Zavattaro, G.: A theory for strong service compliance. In: Proceedings of 9th International Conference on Coordination Models and Languages (Coordination'07). Number 4467 in LNCS (2007) 96–112
15. Baldoni, M., Baroglio, C., Chopra, A.K., Desai, N., Patti, V., Singh, M.P.: Choice, interoperability, and conformance in interaction protocols and service choreographies. In: Proceedings of the 9th International Conference on Autonomous Agents and Multiagent Systems. (2009)
16. Singh, M.P.: Semantical considerations on dialectical and practical commitments. In: Proceedings of the 23rd Conference on Artificial Intelligence. (2008) 176–181
17. Singh, M.P., Chopra, A.K.: Correctness properties for multiagent systems. In: Proceedings of the Sixth Workshop on Declarative Agent Languages and Technologies. (2009) To appear.
18. Schiper, A., Birman, K., Stephenson, P.: Lightweight causal and atomic group multicast. ACM Transactions on Computer Systems **9**(3) (1991) 272–314

19. Dastani, M.: 2APL: A practical agent programming language. Autonomous Agents and Multi-Agent Systems **16**(3) (2008) 214–248
20. Baldoni, M., Boella, G., Genovese, V., Grenna, R., van der Torre, L.: How to program organizations and roles in the jade framework. In: Multiagent System Technologies. Volume 5244 of LNCS., Springer (2008) 25–36
21. Zambonelli, F., Jennings, N.R., Wooldridge, M.: Developing multiagent systems: The Gaia methodology. ACM Transactions on Software Engineering Methodology **12**(3) (2003) 317–370
22. Padgham, L., Winikoff, M.: Prometheus: A practical agent-oriented methodology. In Henderson-Sellers, B., Giorgini, P., eds.: Agent-Oriented Methodologies. Idea Group, Hershey, PA (2005) 107–135
23. Garcia-Ojeda, J.C., DeLoach, S.A., Robby, Oyenan, W.H., Valenzuela, J.: O-MaSE: A customizable approach to developing multiagent processes. In: Proceedings of the 8th International Workshop on Agent Oriented Software Engineering (AOSE 2007). (2007)
24. Alberti, M., Chesani, F., Gavanelli, M., Lamma, E., Mello, P., Torroni, P.: Verifiable agent interaction in abductive logic programming: the SCIFF framework. ACM Transactions on Computational Logic **9**(4) (2008)
25. Bresciani, P., Perini, A., Giorgini, P., Giunchiglia, F., Mylopoulos, J.: Tropos: An agent-oriented software development methodology. Autonomous Agents and Multi-Agent Systems **8**(3) (2004) 203–236
26. Dastani, M., Arbab, F., de Boer, F.S.: Coordination and composition in multi-agent systems. In: Proceedings of the 4rd International Joint Conference on Autonomous Agents and Multiagent Systems (AAMAS), ACM (2005) 439–446
27. Fornara, N., Viganò, F., Colombetti, M.: Agent communication and artificial institutions. Autonomous Agents and Multi-Agent Systems **14**(2) (2007) 121–142