Computing and Informatics, Vol. 34, 2015, 138–166

EXTENDING THE COMMUNICATION CAPABILITIES OF AGENTS

Kai Jander, Lars Braubach, Alexander Pokahr

Distributed Systems and Information Systems Group
University of Hamburg
Vogt-Kölln-Straße 30
22527 Hamburg
Germany
e-mail: {jander, braubach, pokahr}@informatik.uni-hamburg.de

Abstract. Agent technology is in principle well suited for realizing various kinds of distributed systems, but in practice agents are seldomly chosen for realizing real-world applications. One reason hindering agents being used in practice is their cumbersome communication mechanism focused on speech act based message exchange which makes them hard for practitioners used to work in an object oriented way. To broaden the application spectrum of agent technology in practice and make them more accessible for object-oriented developers, this paper presents additional communication means for agents. First, it will be shown how agents can interact using strongly typed service interfaces resorting to asynchronous future based methods. These allow keeping agents autonomous and further support several recurrent interaction patterns within one method call, i.e. without having to use complex message protocols. Second, an extension for binary data streaming via virtual connections will be presented. Its usage resembles established input and output streaming APIs and lets developers transfer data between agents in the same simple way as e.g. a file is written to hard disk. Furthermore, virtual connections allow failure tolerant transmission by multiplexing data across different physical connections. Usefulness of the extensions will be further explained with a real-word example application from the area of business intelligence workflows.

Keywords: Distributed systems, agents, agent communication, streaming, bulk transfer, futures

1 INTRODUCTION

The design and implementation of distributed applications is a difficult task due to many inherent characteristics such as unreliable communication channels and heterogeneity on all hardware and software layers [15]. Programming approaches like service oriented computing (SOA) [23] and multi-agent systems [32] that aim to make the construction of such systems easier by providing both a conceptual model for designing systems and additionally an implementation infrastructure that abstracts away hardware and operating system dependencies. SOA proposes a decomposition of applications in services with clearly defined business interfaces that are often orchestrated by workflows representing company business processes. On a more technical level web services allow for bridging technology heterogeneities and thus foster system interoperability. Multi-agent systems involve the collaboration or competition of multiple software agents in order to achieve a particular goal in the overall system [32]. In constrast to SOA, in multi-agent systems an agent represents an autonomous agent that may act as service provider as well as service user. The population of agents is considered not as static as in SOA and it is a common case that agents are created and terminated during execution of an application. This makes a multi-agent system well suited for application domains that require runtime dynamics and runtime adaptations in response to environmental changes. Despite these conceptual advantages multi-agent systems are currently seldomly choosen for designing and implementing real world applications due to several reasons. One important aspect hampering a direct industry adoption is the complicated communication abstraction that agents possess, as explained next.

The standard approach for inter-agent communication is the direct exchange of messages between agents [19], where the agent platform provides a messaging service as a communication channel and agents submit messages which are then transported to the receiving agent for processing. While this approach works reasonably well for simple interactions between agents, it has a number of drawbacks which delay both the implementation of highly complex systems and impedes its acceptance by mainstream software development:

- If communication consists of more than a simple notification and the other agent is expected to respond, the exchanged messages must include a unique conversation identifier for the conversation instance as well as context information retained by both communication partners about the conversation, such as the state of conversation. Since this information has to be managed at the application level, but actually represents technical information, it adds an additional and unnecessary burden on the developer of such a system.
- Message passing, while occasionally used, is not a common concept used in mainstream object-oriented programming. Software developers are often more familiar with object-oriented method calls. While this does not preclude the use of message passing by more experienced developers, it requires extra effort to

adopt the concepts and results in a steeper learning curve especially for novice programmers.

- While the asynchronous exchange of messages is a very powerful concept with great flexibility, it is also a fairly primitive building block. As a result, developing complex protocols for agents is a very complicated task that needs a careful analysis of the states of involved agents, to interleave messages and investigation of a large number of corner cases. While this can be partially attributed to the asynchronous and distributed nature of the conversation, each message only contributes with a very small part to the conversation, resulting in slow progress and increasing the chance of mistakes. As a result, developers tend to avoid the development of new protocols by employing pre-developed protocols, thus negating the advantage of individual messages as building blocks.

To overcome these difficulties and facilitate usage of multi-agent abstractions in domains with dynamics and adaptation requirements, two additional communication means for agents are proposed. The two alternative options are technically based on a message passing system for inter-agent communication that align more closely with mainstream functional and object-oriented approaches with the goal of reducing overall overhead and easing the learning curve of developers with only a marginal reduction in versatility. With both options, the asynchronous nature of the communication and thus the autonomy of the agents is retained, while the use of message passing is still available when fine-grained control over the communication is required. The first option introduced is an approach based on service calls making SOA abstraction available in multi-agent systems. The second alternative is to allow on-demand streaming of bulk data, which will be presented following the introduction of the first approach.

1.1 Service Calls

Method calls are a well-established approach for passing input values to an instantiated object using arguments and receiving one or more return values once the execution of the method was finished. This approach has also been extended in a variety of ways to support remote execution either as *remote method invocation (RMI)* [22] or *remote procedure calls* (RPC) [29]. Remote method calls like RMI are fairly easy to understand for a developer familiar with local method calls; they both offer input arguments and output return values and would thus be an easy-to-understand alternative to message passing. However, due to the distributed nature and autonomy requirements of the agent, a number of requirements have to be considered:

Services and method declaration: The approach should take the form similar to function calls or object-oriented method calls to reduce the learning curve for the developer and deliver a built-in return path for result values. The agents must also be able to declare available methods and make them available for

other agents to call them. In order to maintain autonomy, the agent must be able to make decisions regarding the call, such as rejecting a request.

Type Safety: In order to reduce errors, the argument and return value should be strongly-typed. Ideally, this should be done through an appropriate programming language construct and employ the typing system of the programming language in order to maximize both familiarity and integration.

Location Transparency: The invocation of the services should be the same regardless of the location of the agent.

Single-threaded agents and asynchronous invocation: In order to simplify and ease the development of individual agents, the approach has to assume that only a single execution thread may be active within a given agent at a single point in time. This reduces the burden on the developer for maintaining data consistency within the agent while still allowing parallel execution of the overall system by employing multiple agent instances. However, communication in a distributed system may have high latency and invocations may therefore require some time before completion. This is magnified if the call invokes a complex operation, which introduces further delay on top of the communication latency. To suspend an agent while performing a remote call to another agent would both undermine the autonomy of the calling agent and introduce the risk of deadlocks via a call loop. The approach must therefore allow asynchronous operation.[1]

Maintaining agent autonomy: Since agents are supposed to exhibit autonomous behavior, they must be able to make decisions regarding the incoming service calls and should be able to refuse a service call based on their own evaluation.

In the following sections a concept for service calls is presented which intends to fulfill the above requirements. The concept is based on services offered by agents and asynchronous service calls.

1.2 Bulk Data Streaming

Although multi-agent systems provide concepts for realizing various kinds of distributed systems, applications with a data centered background are not well supported due to the focus on high-level messaging among agents. In order to build applications that need to transfer huge amounts of binary data between agents two different approaches are available. First, one can directly employ communication libraries e.g. TCP streams. This has the disadvantages of being forced to handle lower-level and in many cases intricate communication aspects at the application level. Second, one can rely on the existing message based communication mechanisms of agents and use them to transfer binary data. The primary problem of

[1] Please note that treating agents as single threaded entities is not a conceptual drawback and is done by most available agent platforms. The main idea behind this design decision is that it introduces an agent as concurrency concept, i.e. changing the number of agents allows scaling the concurrency in the system.

this approach is that it will not work with arbitrary large data as it cannot be held in main memory completely, and additionally, performance degradation is likely to occur. In connection with multi-agent systems the following requirements need to be considered carefully:

Location transparent addressing: Addressing should be done between agents and should be location transparent, i.e. it should be possible to transmit data between agents without knowing their location.

Infrastructure traversal: Data transfer must be able to cope with the existing infrastructure characteristics and restrictions. This means that e.g. firewall settings might constrain the ability to open new connections for transmissions. As a result, existing communication channels have to be reused and shared.

Failsafe connections and heterogeneous multihoming: Data transfer between agents should be as failsafe as possible and use all available means to reach the other agent, for example during connection breakdowns etc. Multihoming requires that agents may use different connections to transfer data of the same stream. Furthermore, in case of big files it is crucial to avoid complete retransmissions if parts already have been successfully transferred. This definition of multihoming is analogous to the definition used in other network types such as IP networks [12].

Non-functional properties: The quality of service characteristics such as non-functional properties of the transfer should be configurable. Important properties include e.g. security, reliability, and priorities.

In Section 2, an example scenario is introduced which will be used as a running example to explain the communication concepts and demonstrate their usefulness in a real world scenario. In the third part of this paper an approach will be presented that addresses these requirements with a distributed streaming concept based on virtual connections. The remainder of this paper is structured as follows. Section 4 presents the approach itself and its implementation. Thereafter, Section 5 illustrates the usage of the approach within the previously introduced real-world example scenario. Section 6 compares the proposed approach to existing solutions for both the service call and the streaming concept and Section 7 gives some concluding remarks and a short outlook over planned future work.

2 EXAMPLE APPLICATION SCENARIO

The design choices and implementation of the communication options will be evaluated based on an ongoing commercial project called DiMaProFi (Distributed Management of Processes and Files), which deals with specific challenges in the area of Business Intelligence. The primary objective of the project is the implementation of a distributed workflow management system for extraction, transformation and loading (ETL) processes [25]. These processes pre-process and transform and extract information from very large repositories of data which is distributed throughout

multiple locations within a data warehouse. Since the amounts of data are considerable, moving them to a central location for processing is inefficient, which makes certain processes impractical to implement. Instead, the goal is to process and condense the data directly at the location where they are stored before they are loaded for further inspection by domain experts. The user can graphically design and then execute processes which employ remote resources to accomplish of a variety of task throughout the data warehouse.

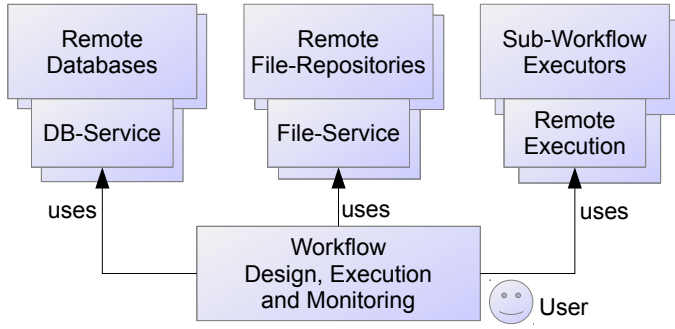


Figure 1. DiMaProFi usage example

The user can thus perform a variety of remote task on remote nodes as shown in Figure 1. In addition, sub-workflows can be started on different nodes to enhance performance and workflow reuse. Remote transaction performed by the nodes typically include operations like file copying and compression, remote file transfer, database transactions and e-mail services. Workflows can be monitored while they are executing or their execution log can be reviewed after execution has finished. This allows the user to evaluate the performance of the workflows and intervene if issues occur.

Since the workflows differ depending on the customer, and data may be changed due to business decision and reorganization, the system must allow rapid and easy implementation of a plethora of highly variable workflows of this type, which then must be able to execute the transactions remotely and in parallel, if possible. In the following, the proposed communication concepts are introduced. The DiMaProFi application will be used as a running example to explain the communication concepts and to demonstrate how these concepts can be used to support this type of ETL workflows with the implementation of such a workflow system in elegant and efficient manner. The application itself will be further elaborated in Section 5.

3 SERVICE CALL APPROACH

Based on the requirements for a service call-based communication approach a number of software engineering techniques have been reviewed and considered as possible

solutions, where the most important aspect is a close integration with object-oriented concepts and ease of use. However, compromises had to be made and as a result, the following possible solutions have been developed to address the requirements:

Services and method declaration: This requires the agents to declare services and individual method calls. This could be achieved through two different approaches: declaration using elements of the programming language such as interface declarations and declaration using a specially-defined language. Since a special-purpose language would require the translation between data coming from the primary programming language and flowing into the service call system, type safety would be difficult to maintain and calling services would be unlike calling other methods in the primary programming language. As such, the first approach was preferred. First, the concept and implementation is introduced followed by usage examples. Lastly, related work is presented, followed by a general conclusion together with the bulk streaming concept.

Type Safety: Since services are to be declared as interfaces in the primary programming language, the language compiler will ensure the type safety of the service calls. If the developer attempts to pass an invalid argument into a service call or retrieve a wrongly-typed return value from the call, the compiler will flag the problem and refuse to compile the agent until the developer has resolved the problem.

Location Transparency: Services may be offered by agents that are running on remote platforms or hosts, which means that the local thread executing the program cannot simply execute the code of the remote agent. However, systems for remote method invocation offer a solution for this problem by offering the local agent a proxy object for the remotely called object which will then handle the marshalling and messaging to the remote location.

Single-threaded agents and asynchronous invocation: The basic method call concept as it is available in most object-oriented programming language is a synchronous operation. From the perspective of the caller, execution is halted, the functionality of the method call is performed and the execution continues only after the method call has finished and the return values are available. As a result, this approach is fundamentally incompatible with asynchronous operation since this does not allow halting the calling. This problem has generally been solved by employing futures with a callback. In addition, another approach is possible using thread suspension and thread management, which will be discussed in more details in the implementation section.

Maintaining agent autonomy: This requirement can be fulfilled using two options. First, an interceptor chain is used to pre-empt service calls and give the receiving agent an opportunity to reject the call. Alternatively, the agent may simply return an exception instead of the return values.

These design decisions lead to a novel approach of interacting entities, which is not conceptually equivalent to a classical distributed object-oriented systems employing

RMI. One important difference is that entities remain autonomous despite the fact that they act as service providers and offer object-oriented corresponding service interfaces. In contrast to classical objects the entities do not execute service method invocations directly, but instead, use the notion of a service invocation task that is internally scheduled in the agent. The agent can then decide if and when it wants to execute the request and send back the results. Key advantage of this approach is that it allows designing large-scale systems using service oriented architectures with strongly typed software technical interfaces. Using purely agent oriented technology one could also design a services based agent system, but at the implementation layer one would have to resort to string based service descriptions, messages and protocols for realizing the functionalities.

3.1 Service Call Architecture

Based on the requirements, the methods offered for communication between agents should be represented as services implementing the individual methods as part of a service interface. This service interface is then declared using the programming language.

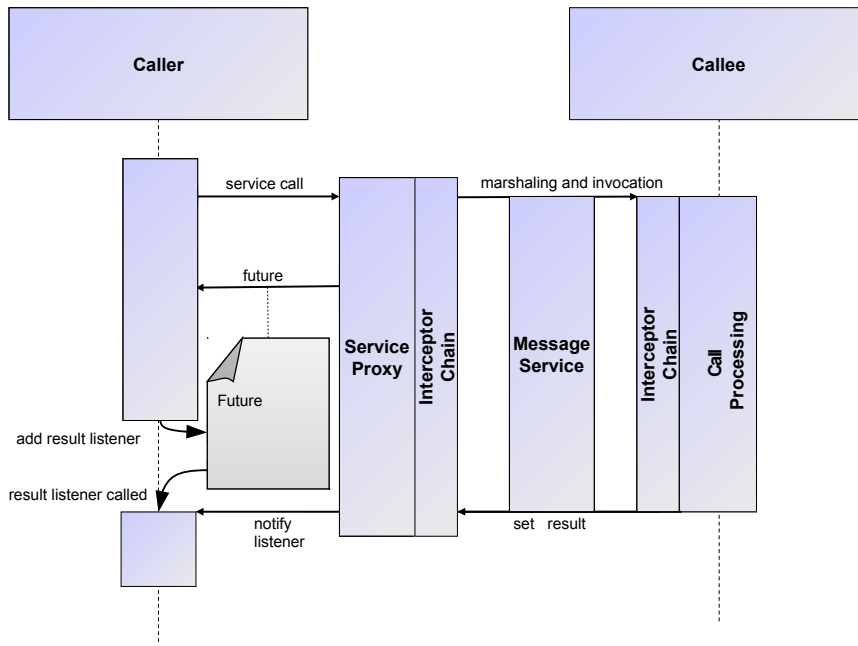


Figure 2. Service call architecture

The system will then provide a service proxy to the caller implementing the service interface and providing the agent with a proxy object implementing the service methods (see Figure 2). It has been shown as part of previous work that using method calls does not necessarily impede the autonomy of agents, if done properly [8]. The call is processed by an interceptor chain and its parameters are then serialized and transmitted using the message service of the platform. On the receiving side, the message is decoded and the call is passed into the receiving interceptor chain. An action in the form of an executable step then performs the actual call within the receiving agent, which can return either a result value or an exception, which is then passed back to the calling agent using the same approach.

This approach trivially allows the implementation of request-type protocol similar to the FIPA Request Interaction protocol [20]. However, more complex protocols can be implemented either by implementing services in all participants and chaining multiple service calls or by offering special futures which allow more complex interaction than the reception of a return value.

The next section will describe how to declare service interfaces and how to make use of the future concept to handle asynchronous service calls. This will be further expanded upon with an example of a more complex type of future for specialized tasks.

3.2 Service Interfaces and Futures

Agents within the system are able to declare services, each containing potentially multiple methods that can be called. Services are defined in two parts: First, a Java interface declares the signature of the methods the service offers. This interface will be visible to outside agents using the service and define the method name, parameters and return value. The second part consists of a concrete implementation of the service as a separate Java class implementing the service interface and containing the code the agent executes if one of the service methods is called.

```
public interface IFileCompressionService
{
  public IFuture⟨Void⟩ compress(String inputfilename, String outputfilename);
  public ISubscriptionIntermediateFuture⟨LogEntry⟩ subscribeToLog();
}
```

Figure 3. File compression service interface

Figure 3 illustrates how a service interface can be declared similarly to the file compression service used in DiMaProFi[2]. Both declared methods offer a future as the return value. This future is returned instantly as expected from a regular object-oriented method call, but merely represents the promise to deliver the actual

[2] The interface has been reduced and simplified both for readability and business secrecy reasons.

return value asynchronously at a later point in time [28]. This means that a future is a placeholder object in which the callee will push the result value of the call as soon as it is available. In the interface different future types are used to render them asynchronous. The example demostrates that multiple types of futures may be returned by the call which represent different kinds of functionality regarding the return value.

The basic future offered by the *compress()* method will deliver a single return value at a later point. In this case the return value is typed as a Void value, which means only a null value is eventually returned, merely indicating the successful compression of the remote file. However, any bean-conform Java object may be returned by this type of future. The actual returned value can be retrieved by the caller using two approaches.

```
future.addResultListener(new IResultListener<Void>()
{
  public void resultAvailable(Void result)
  {
    // Process result
  }
  fulfills the requirements and
  public void exceptionOccurred(Exception exception)
  {
    // Handle exception
  }
}
```

Figure 4. Retrieving the service call return value using a listener as a callback

First, the caller can add a listener to the future as a callback as shown in Figure 4. Once the remote call is processed and returned, an action is scheduled on the calling agent to invoke the appropriate listener method with the result of the call. While this method is asynchronous and allows the agent to perform independent actions while waiting for the call to return, it can be inconvenient for the developer since the code is no longer sequential: The execution continues immediately after the *addResultListener()* method has been called and the methods of the inner listener class are called at a later point, which may be confusing to the developer. This can be avoided by employing the second approach for retrieving the result value:

```
Void result = future.get();
```

This simpler approach for retrieving the return value by simply invoking the *get()* method of the future. The method will suspend the calling thread, but in order to avoid stalling, the agent injects a second thread into the agent which will serve as the single agent thread until the service call returns. Once the return value is available, the second thread is removed from the agent between actions while

the second thread is not active within the agent and the original thread is allowed to continue with the result value. This approach both ensures that only a single thread is active within the agent at any given time, avoiding multithreading issues as well as allowing the service call to work in a synchronous fashion as expected by an object-oriented developer.

3.3 Complex Protocols and Futures

As shown in Figure 3 in the previous section, service calls may return specialized futures instead of the default futures, which implement more complex behavior. For example, the *subscribeToLog()* method is to provide periodic updates of the compression log in which file compressions will be recorded. Since the default future can only return a single value, using it to implement such a subscription functionality would be fairly complex. The subscribing agent would also have to offer a service through which it can receive the subscription updates.

```
future.addResultListener(new IntermediateDefaultResultListener⟨LogEntry⟩()
{
  public void intermediateResultAvailable(LogEntry result)
  {
    // Process log entry
  }
}
```

Figure 5. Adding an intermediate result listener to a subscription future

Instead, a newly devised ISubscriptionIntermediateFuture offers a better solution. The subscribing agent simply calls the subscribe method and receives the aforementioned future. This future allows the subscribing agent to add an intermediate result listener (see Figure 5), which, unlike the default result listener, is able to receive multiple intermediate results. Alternatively, the caller may also periodically poll for new subscription updates using a dedicated method offered by the future:

```
LogEntry update = future.getNextIntermediateResult();
```

If the subscriber no longer wishes to receive updates, the agent can cancel the subscription by simply calling the *terminate()* method of the subscription future:

```
LogEntryupdate = future.terminate();
```

This results in the subscription being canceled and the listener no longer receiving any updates from the subscription. The subscription future therefore allows a developer to implement a fairly complex protocol similar to the FIPA subscription protocol [21], which deals with situations where an agent offers periodically updated information to other agents interested in the information.

As this example shows, the service call approach allows the implementation of more complex protocols through the use of specialized and complex futures. This allows the developer to use the protocols in a simple and straightforward manner which does not greatly diverge from an object-oriented approach.

3.4 Evaluation

	# Messages Types	# Phases	# Files	Lines of Code
request	6	3	3	546
subscribe	6	3	5	406
dutch auction	7	3	3	896
engl. auction	7	4	3	993
canel meta	4	2	4	403
(iter.) contract net	16	4/(iter.) 5	5	1 526

	# Interfaces	# Methods
request	1	1
subscribe	1	1
dutch auction	1–2	n/a
engl. auction	1–2	n/a
canel meta	0	0
(iter.) contract net	1–2	3

Figure 6. Comparison of interaction protocol and service complexity

The quantitative evaluation of the service interface approach compared to the agent interaction protocol mechanism is difficult to conduct, because it deals with development complexity and efficiency which cannot be easily measured. In order to deal with this problem in the following multiple indicators of complexity have been used. Regarding interaction protocols the complexity is assessed using simple design and implementation metrics. On the design level the number of different message types and the number of protocol phases are considered, whereby the latter aspect describes how often initiative changes between initiator and participant sides. Furthermore, the implementation conplexity has been measured using the generic protocol implementation as part of the Jadex agent framework [10]. In this respect the number of implementation files as well as the lines of code accumulated over all files have been captured. None of these metrics is directly applicable to the service approach so that here the number of interfaces as well as the number of methods have been counted.

The evaluation results taking into account several FIPA standard interaction protocols are shown in Figure 6. It can be seen that the number of different messages varies between four and sixteen and that the number of phases increases in the same protocols already having a large message type complexity. The implementation details are not as accurate as the design metrics, but in general support them

and show that auctions and (iterated) contract net are among the most complex protocols. An interesting point is that even the basic and quite simple FIPA request protocol [20] has been devised with six different message types and three phases while it can be mapped to one method call.

It has to be noted that many design choices exist when coping with more complex protocols like auctions and contract net. One important aspect concerns which side should be equipped with a service interface (initiator, participant or both). This depends, to some extent, on the concrete application demands and is determined by the number of necessary phase shifts of the protocol. Considering the request protocol, the original number of three phases in the protocol representing request, agreement or refuse and result notification can be pushed into one service with one method. The request phase corresponds to the invocation of the method and the result notification is handled to the return value. Moreover, if the participant wants to refuse the request at any time it can always signal an exception to the caller. This example shows the general phenomenon underlined by the results in Figure 6 that methods are more complex building blocks than messages and allow abstracting protocols in a more compact way. While the design of more complex protocols is not completely solved, due to an inherent ordering in method calls of interfaces making such kind of service interfaces harder to use, it nevertheless reduces the overall effort for the developer compared to a pure message passing.

4 DATA STREAMING APPROACH

The requirements of the previous section had been carefully analyzed and strongly influenced the design of the streaming architecture presented later. Here, the findings are shortly summarized according to the already introduced categories. As an additional point the agent integration had been added, because the agent characteristics also determine the set of possible solutions.

Location transparent addressing: This implies that a connection should have an agent as a start and endpoint. Furthermore, the streaming mechanism should be enabled to use the existing agent platform addressing to locate the target agent platform.

Infrastructure traversal: In order to cope with different environments and security settings, the solution use existing communication channels for multiple streams, i.e. multiplex the data.

Failsafe connections and heterogeneous multihoming: Failsafe connections require that streams should be able to communicate via different underlying transport connections, i.e. the mechanism must be able dynamically switch in case of a breakdown. Moreover, the required intelligent usage of underlying transports requires a layered approach in which an upper coordination layer selecting and managing the underlying transports.

Non-functional properties: The coordination layer has to consider the properties when selecting among different transport options (e.g. whether a transport is encrypted, authenticated etc.)

Agent integration: The streaming mechanism should be accessible to the agents in a non-disruptive way, i.e. streams should be an option in addition to the traditional message sending approach.

4.1 General Stream Architecture

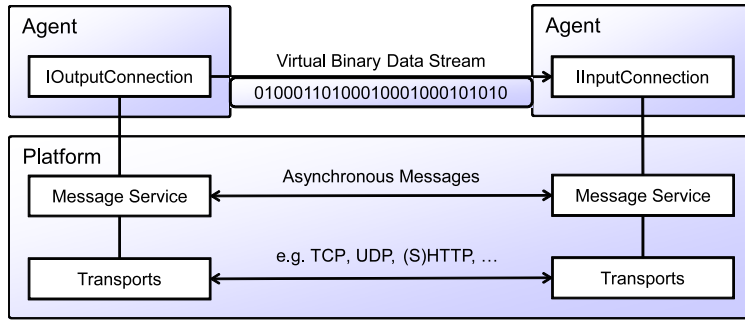


Figure 7. Stream architecture

The streaming architecture originally proposed in [11] is depicted in Figure 7. From a high-level view an agent should be enabled to directly use input and output connection interfaces – in addition to sending messages – to directly stream binary data to/or receive data from another agent. The figure also shows that the basic envisioned architecture relies on the standardized FIPA platform architecture [18] in a sense that it is assumed that on each agent platform a MTP (message transport protocol) exists that is capable of sending asynchronous messages to agents of the same and other platforms. For this purpose it uses different transports, which utilize the existing communication technologies such as TCP, UDP or HTTP to transfer the data.

4.2 Stream Usage

In order to better understand the envisioned usage from an agent perspective, the important streaming interfaces are shown in Figure 8. Each connection (*IConnection*) has a connection ID as well as two endpoints, an initiator (agent) and a participant (agent). Each side is free to close the stream unilaterally at any point in time. The other side will be notified of the termination via a corresponding exception. Furthermore, each connection may be initialized with non-functional properties consisting of key value pairs.

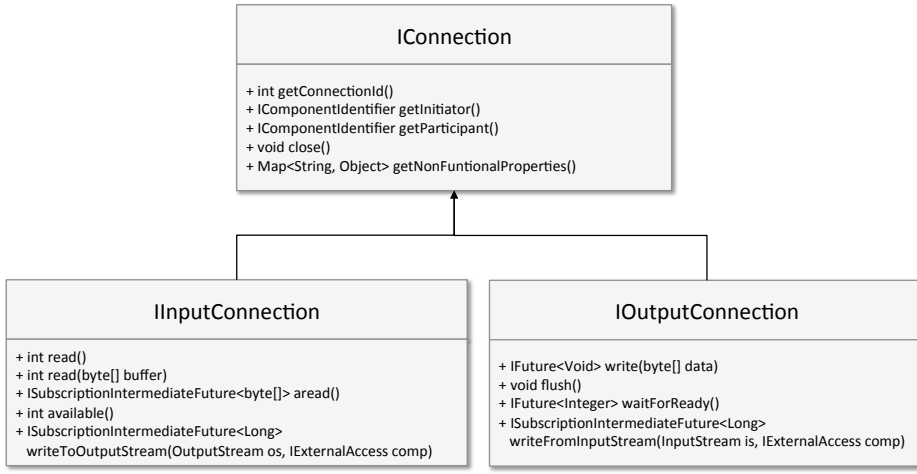


Figure 8. Stream Interfaces

An output connection (*IOutputConnection*) is used to write binary data in chunks to the stream (*write*). First, the concept and implementation is introduced, followed by usage examples. Lastly, related work is presented, followed by a general conclusion together with the bulk streaming concept. As it is often the case, the sender and receiver cannot process the stream data at the same speed, a new mechanism has been introduced to inform the output side when the input side is ready to process more data (*waitForReady*). Finally, also a convenience method has been introduced that allows automatical processing the Java input stream by reading data from it and writing it into the output connection until no more data is available (*writeFromInputStream*).

The input connection (*IInputConnection*) offers methods to read data from the stream. These methods include variants for reading a single byte, as well as a complete buffer of bytes. Before calling these methods it can be checked how much data is currently available at the stream (*available*). Moreover, it is possible to register a callback at the stream and automatically get notified when new data is available (*aread*). The input connection also possesses a method for connecting to standard Java streams. In this respect, the input connection allows automatical writing all incoming data to a Java output stream (*writeToOutputStream*).[3]

[3] Please note that in contrast to Java streams all connection interfaces are non-blocking using future return values, although the method signatures look similar otherwise. Blocking APIs are not well suited to work with agents as these are expected to execute in small steps to remain reactive. An agent that would directly use a blocking stream method could not respond to other incoming requests while it waits for the blocked call to return.

4.3 Low-Level API

Besides the functionality an agent uses to send and receive data from the stream the question arises how streams are created by the initiator and received by the participant of a connection. For the first part the interface of the message service has been extended with two methods that allow for creating virtual connections to other agents. The method signatures are shown in Figure 9. The caller is required to provide the component (i.e. agent) identifier of the initiator and the participant of the connection. Furthermore, optionally additional non-functional properties can be specified which have to be safeguarded by the message service during the stream's lifetime. As a result of the call the corresponding connection instance is returned.

An agent that is used as a participant in one of the create connection methods is notified about the new connection via the hosting platform. This is done via a new agent callback method (*streamArrived*) that is automatically invoked whenever a new stream is created. Behind the scenes the platform of the initiator contacts the platform of the participant and creates the other end of the connection at the target side. This connection is afterwards passed as a parameter to the streamArrived method call. Having received such a callback the receiving agent is free to use it as it deems appropriate. Of course, it can also do nothing and ignore such incoming stream connection attempts.

```
IFuture⟨IOutputConnection⟩ createOutputConnection(IComponentIdentifier initiator,
  IComponentIdentifier participant, Map⟨String, Object⟩ nonfunc);

IFuture⟨IInputConnection⟩ createInputConnection(IComponentIdentifier initiator,
  IComponentIdentifier participant, Map⟨String, Object⟩ nonfunc);
```

Figure 9. Extended message service interface

4.4 High-Level API

For active components [9], which in brief are extended agents that can expose object oriented service interfaces, another more high-level API has additionally been conceived. As interactions with active components are primarily based on object-oriented service calls, it becomes desirable to be able to use streams also as parameters in these service calls. Using the high-level API an active component can declare streams as arbitrary input parameter or as the return value of a call. This allows passing a stream directly to another agent solely by calling a service method.

Realization is complicated by the fact that method signatures contain the expected connection type of the callee but not of the caller. This means that a caller that wants to stream data to the callee has to create an output connection and write data to it but has to pass an input connection as parameter to the service call for the callee to be able to pull the data out of the stream. To solve this issue new

service connection types have been introduced, (*ServiceInput-* and *ServiceOutput-Connection*) which allow fetching the corresponding opposite connection endpoint (*getOutputConnection()* on ServiceInputConnection and vice versa).

Support of non-functional properties has also been mapped to the high-level API. As these aspects should not be part of method signatures, that are meant to be functional descriptions, an annotation based approach has been chosen. For each supported non-functional property a corresponding Java annotation exists that can be added to the method signature of a service, i.e. *@SecureTransmission* can be used to ensure an encrypted data transmission.

4.5 Implementation Aspects

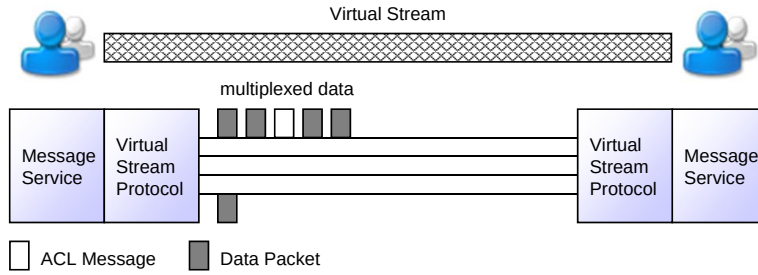


Figure 10. Stream implementation view

A high-level view of a virtual stream between two agents as end users is shown in Figure 10. It can be seen that, even though the agents perceive the stream as one logical connection, internally, it relies on a virtual stream protocol that may use different transport protocols to the target. The general transfer mechanism generates small sized packets from the incoming stream data, numbers them consecutively and uses the transports that best match required QoS to transfer them. In this way the data packets of the stream can be multiplexed with other ordinary agent ACL messages and may also arrive out of order at the destination.

The implementation distinguishes different responsibilities via different layers (cf. Figure 7) and has been implemented as part of the Jadex platform [9]. On the top layer, the input and output connections ensure that streams comply with the functional and non-functional stream requirements. These requirements are addressed in the virtual stream control protocol, which is based on well-established TCP concepts [24].[4] Details of the implementation of the connections as well as of the protocol are depicted in Figure 11.

[4] A virtual connection has to provide the requested service guarantees regardless of the existing infrastructure and underlying communication stack. For this reason it is necessary to reconstruct many aspects of TCP and other protocols on the upper layer.

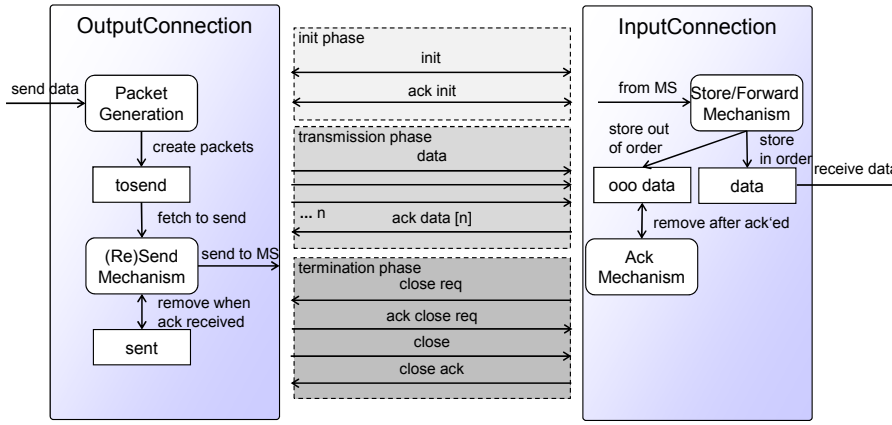


Figure 11. Connection and protocol implementation

The connection protocol consists of three different phases. The init phase is started by the initiator side (which can be an input or output connection) by sending an init message to the participant side. The connection is established when the corresponding ack message is received by the initiator side. In the transmission phase data packets are passed from the output to the input connection using bulk acknowledgement messages. Finally, in the third phase connection teardown is handled via specific handshake messages. In case the output side wants to shutdown the connection it just sends a close message that needs to be acknowledged by the input side. If, on the other hand, the input side wishes to terminate the connection first, a close request message is sent. In response, the output connection will disallow the application level to push data into the connection and guarantees to deliver all unsent data to the input side before it terminates using the close message.

An output connection sends stream data in form of packets with a fixed size via the underlying message service. Thus packets, provided by the application layer, are created by either joining too small data chunks or by fragmenting larger ones depending on their size (packet generation). Depending on the current state of the connection, created packages are either sent immediately or stored for later sending in an internal buffer (tosend). Once a packet could have been sent, it is removed from this buffer and stored in another one (sent) waiting for its acknowledgement. A resending mechanism keeps track of still unacknowledged packets and is in charge of triggering a resend after a timeout period. Furthermore, the connection realizes flow control by using a sliding window that adapts the sender's connection speed to the speed of the receiver.

The input connection handles incoming packets in the following way. First, a store/forward mechanism determines if a packet is an in or an out of order entity by comparing its internal counter with the received packet number. In case the packet is

in order, it will store the packet in a buffer (data) that represents the available data of the input connection, i.e. the buffer can be accessed directly from the application layer via corresponding read requests. Such read requests will automatically remove the delivered elements from the buffer. In case the packet is out of order it will be stored in an internal buffer (ooo data) and will remain there until the missing intermediate packets have been received. For performance reasons, the acknowledgement mechanism uses a bulk mode and does not send an acknowledgement for each single data message. Instead, it will wait until a specified number of packets has arrived and then acknowledges the range with one message. This mechanism leads to problems when the number of out of order messages increases so that an additional timer based acknowledgement for those has been realized. The input connection receives and collects packets to forward them to the application level in the correct order.

The underlying message service has been extended to manage virtual connections and support sending messages belonging to the virtual connection protocol. Whenever the API is used to create a virtual connection (cf. Figure 9) the message service internally creates a connection state at both connection sides and also starts a lease time based liveness check mechanism to ensure that the other connection side is still reachable. In case the lease times indicate that the connection has been lost it is closed unilaterally. The transport layer itself does not need changes to support streaming.

4.6 Evaluation

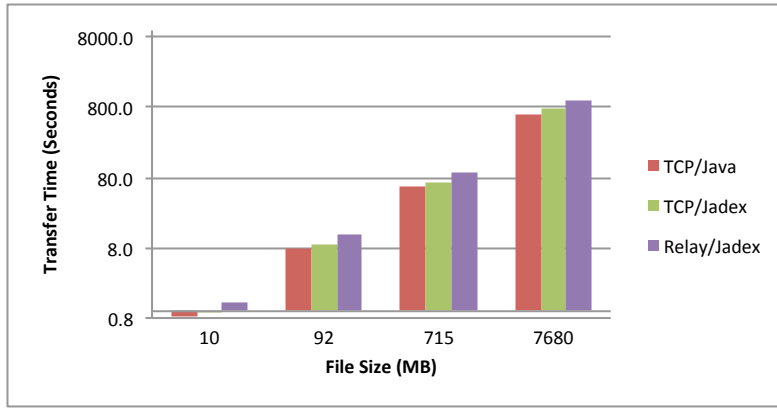


Figure 12. File transfer times of different file sizes (logarithmic scale)

As described above, the stream protocol is realized on top of the existing agent messaging infrastructure, which consists of different message transport protocols. E.g. one transport protocol uses TCP connections for transmitting messages and

another one uses an HTTP-based central relay server for allowing agent communication also behind firewalls. To measure the overhead added by the agent message transport, files of different sizes have been transferred using a direct TCP implementation in Java, as well as the stream implementation running on top of the TCP- and relay-based message transport protocols (cf. Figure 12). The experiments were performed in a closed 100 Mbit/s network and retransmitting the files three times showed no significant deviation in the transmission times (e.g. less than two seconds for all transport protocols when using a file size of 715 MB).

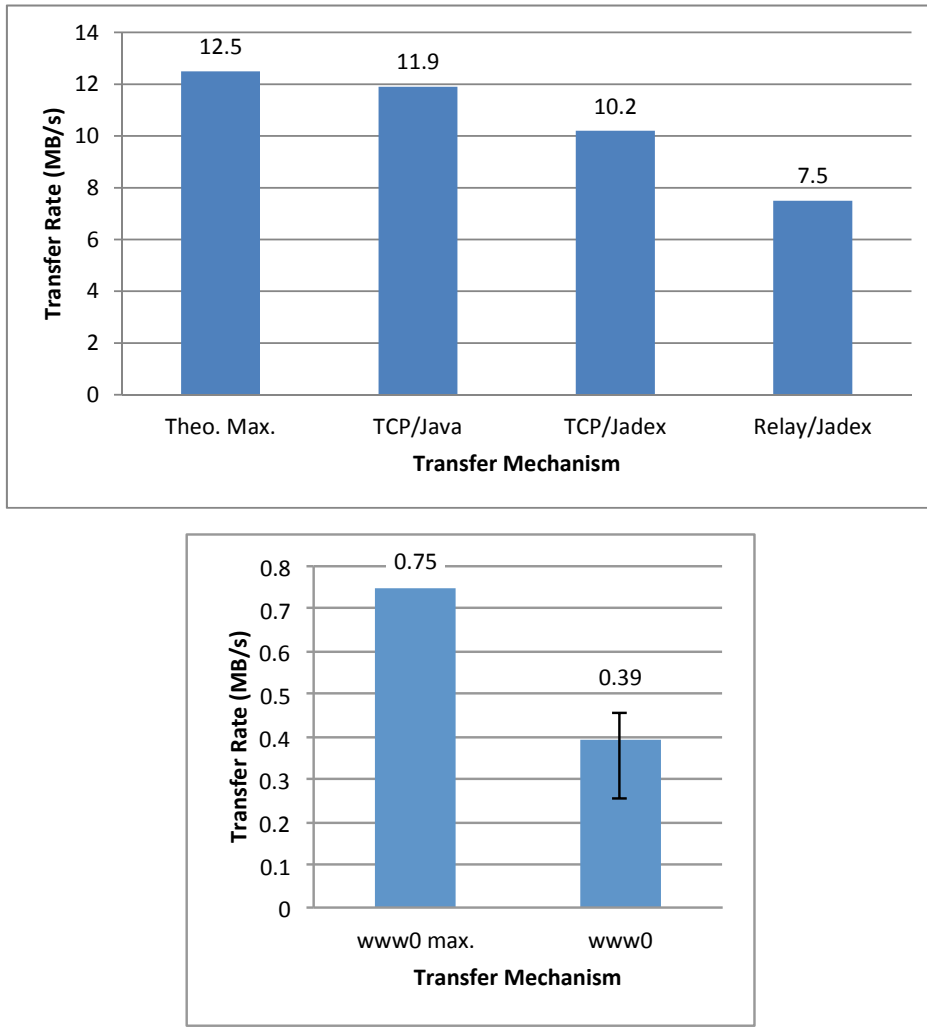


Figure 13. Stream performance in a closed (up) and public (down) network

The results were further analyzed to extract the effective transfer rate with regard to each setting. For comparsion, in Figure 13 (up) also the theoretical maximum of the network (12.5 MByte/s) is shown. It can be seen that the *TCP/Java* implementation is very close to this theoretical maximum. The stream implementation on top of the TCP message transport (*TCP/Jadex*) naturally adds some overhead, but still achieves 83 % of the performance of the pure TCP connection. When using the relay transport with an intermediate node (*Relay/Jadex*), performance drops to 59 % due to additional overhead, caused on the one hand by increased processing time and on the other hand by extra data pertaining to the HTTP protocol. To measure the performance in realistic environments a second set of test has been performed using a public relay server (www0) in a different network connected to the internet with 6 Mbit/s (cf. Figure 13, down). Here the theoretical maximum is at 0.75 MByte/s. The effective performance was measured at an average of 0.39 MByte/s with considerable variation from 0.25 MByte/s to 0.45 MByte/s as shown in the error bars of the rightmost bar. One key reason for the variation is the differing load of the relay server, which is concurrently accessed by the public.

The results of the evaluation show that the stream protocol itself adds only minimal overhead to the existing message transport protocols, given that even with the messaging overhead included it still achieves 83 % of pure TCP performance. Using alternative transports such as the relay, allows flexible usage scenarios also in realistic settings, such as public networks.

5 REAL WORLD APPLICATION

In this section the streaming approach is further explained by dint of the already introduced DiMaProFi workflow management project with Uniique AG. Customer-specific ETL processes are generally based on files which need to be loaded, transformed and then written into the customer's data warehouse. As an example a simplified version of a real world ETL banking process is used in the following. Here, source files are deposited in a special folder monitored by a process on a file server. Since the file sizes are considerable and the ETL process requires a substantial amount of processing time, the transformation processes are executed on different machines in the network in parallel for increased performance. The file server and the data warehouse are separated by a firewall which allows only certain traffic to pass.

Figure 14 shows an example for such a process and Figure 15 illustrates a corresponding infrastructure layout in which such a process is executed. Whenever a customer file is stored on the file server, the monitoring process is notified. It initiates the ETL process on a remote process server that is separated via a firewall from the file server, i.e. both servers can only communicate via a manually arranged platform communication channel. The process requests the binary stream from the file server (fetch customer file) and the data stream is delivered via the already present platform communication channel. In this scenario this is crucial

as no extra TCP connection can be opened due to the firewall restrictions. After transmission the file is stored in a temporary folder on the target machine. Then the received data is cleaned up with respect to the contained address data and thereafter two parallel transformations are performed on the same output data via external worker machines. This task delegation is realized in the process by using a service based approach, i.e. the transformation tasks are designed via service invocations that search for a suitable service via the transformation interface and also rank the results according to the current load of the workers. In this way load balancing is automatically achieved and after processing the resulting data sets are written in parallel into the data warehouse. This process is performed in parallel on multiple machines for each file that has been deposited on the file server.

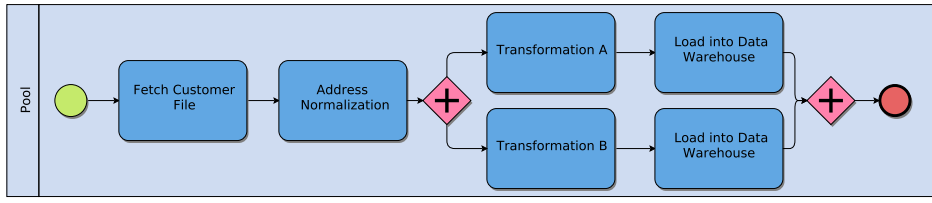


Figure 14. An ETL process loading a file, transforming and writing it to the data warehouse

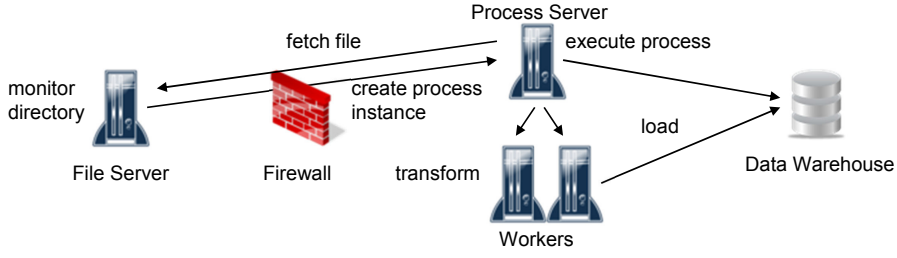


Figure 15. Example infrastructure layout

The code of the fetch customer file task, which uses service-based agent interaction and the high-level streaming API, is depicted in Figure 16. It consists of the (reduced) interface of the file service, which offers a method *fetchFile()* to retrieve a remote file.[5] As a parameter the method takes the local file name and as a result it delivers an input connection that can be used to download the file data. The code for downloading the file is shown below. First, a service provider for fileservice is searched (details omitted) and the input connection is obtained by calling the *fetchFile()* service method of the file server. Afterwards a file output stream for the

[5] The get() method is part of the future API and causes the call to block until the asynchronous invocation has returned.

temporary file is created and the whole file content is automatically written to this file output stream by calling *writeToOutputStream()*. Please note, that this method takes the agent as argument as it executes the stream reading as agent behavior. The *get()* operation blocks until no more data is received and all data has been written to the file. Finally, the stream is closed.

```
// File service interface and method
public interface IFileService {
  public IFuture⟨IInputConnection⟩ fetchFile(String filename);
  ...
}

// Fetch file task excerpt
IFileService fileservice = searchService(IFileService.class);
IInputConnection icon = fileservice.fetchFile().get();
FileOutputStream fos = new FileOutputConnection(tmpfolder+"/"+filename);
icon.writeToOutputStream(fos, agent).get();
fos.close();
```

Figure 16. Code excerpts of fetching a remote file

6 RELATED WORK

Agent interaction is an active field of research and thus many different approaches to agent interaction exist (cf. e.g. [7, 8]). A large group of approaches still relies on low level messages, but offers programming tools to simplify the development of message-based interactions. For example code generators [16, 13, 27] and protocol interpreters [17] offer an easy way from protocol descriptions in, e.g., AUML [2] to behavior that can be executed on top of agent platforms such as JADE [4]. Other approaches try to raise the level by considering, e.g., interaction goals [14, 7], speech-act semantics (e.g. JADE semantic agent [4] and LS/TS SemCom architecture [31]) or commitments [33]. Most interesting with regard to the approach presented in this paper are approaches, that try to bring agent interaction closer to object-oriented programming.

Most prominently, this path is followed by the agents and artifacts (A & A) paradigm [26]. Here, artifacts are introduced as a first-level design concept, which represents tools and items that agents work with. Artifacts offer an object oriented interface to the agents and can be used to mediate between multiple agents as an indirect form of interaction. Therefore the A & A paradigm allows both message-based direct agent interaction as well as OO-style interaction using artifacts. Direct OO-style interaction is realized in AmbientTalk [30], which is not fully agent-oriented, but a framework and programming language for ambient intelligence based on simple actors. AmbientTalk allows method calls on other local or remote actors by providing similar decoupling mechanisms as in our model.

The approach in this paper shows that essential service and OO-method call functionality can be directly bound to agents and in this way facilitates the development of distributed applications further. In the following the second interaction extension regarding streaming is considered further.

	Agent Message Communication	Network Communication	Overlay Networks	
	Jade/JMS	TCP Connection	SpoVNet	RON
Streaming Support	-	+	-	+
Agent Integration	+	-	-	-
Location-transparent Addressing	+	-	+	-
Infrastructure Traversal	-	-	+	+
Heterogeneous Multi-Homing	+	-	+	-
Failsafe Connections	-	-	o	o
Automatic Configuration	-	-	-	-
Non-functional Properties	-	-	+	-

Figure 17. Streaming support requirements and support by different approaches

As mentioned in Section 4, the powerful streaming support includes a number of requirements that are not generally part of agent communication systems and network communication is often used to supplant it. However, overlay networks may offer an approach unrelated to agents that promises to meet some of the requirements. As a result, three basic categories are considered: agent communication, direct network communication and use of overlay networks, examples for each are shown in Figure 17.

Streaming has not been a priority for agent systems. The traditional approach for agent communication centered around the exchange of speech act based messages, e.g. in JADE [3], which typically uses HTTP to transfer messages. Messages free the agents of low-level communication details and provide a form of location-transparent addressing. This approach is suitable for the exchange of small amounts of data, however, the lack of explicit streaming support forces agents to send bulk data in large messages, which can unnecessarily block the agent, or the messaging layer of the agent platform.

It is thus often suggested to use direct network connections such as TCP sockets for streaming and bulk transfer [5]. However, this forgoes the advantage of location-transparent addressing and burdens the agent with a number of low-level tasks, among them networking concerns such as firewall traversal. Furthermore, calls to such communication channels are often blocking, forcing intra-agent multithreading and increasing risks of data loss and deadlocks. In addition, if the connection is interrupted, recovery is difficult and if the chosen protocol like TCP is unavailable, the agent is unable to stream data at all. Both network connection and agent messaging only provide little support for non-functional features. While network connections often have QoS implementations, their configuration is hard and must be done at the system level. Application-level QoS-features such as the IPv4 type-of-service (TOS) field are generally ignored by routers.

An alternative consists in using overlay networks, which often bundle some of the required features such as (heterogeneous) multi-homing, location-transparent addressing and infrastructure traversal. While overlay networks do not provide specific support for agents, they often include a number of useful features. For example, Resilient Overlay Networks (RON) [1] allows streaming by tunneling TCP connections and allows multi-homing and, given an appropriate configuration, infrastructure traversal by relaying communications using other nodes. However, the multi-homing is not heterogeneous and thus connections are only failsafe in a limited sense. Furthermore, the addressing issue is not resolved and non-functional properties are unsupported. The overlay network framework Spontaneous Virtual Networks (SpoVNet) [6] does support both: location-transparent addressing through unique identifiers and specification of non-functional properties. It also provides some means for heterogeneous multi-homing using multiple means provided by underlays to transfer messages. However, it does not provide a streaming support.

In general, overlay networks seems to be the most promising to provide a solution for the requirements, but cover only a subset of the required feature set. Combining a multiple of such networks may be possible; however, this is hampered by problems such as integration of different programming languages.

7 CONCLUSIONS AND OUTLOOK

In this paper a concept and implementation of agent communication using both service calls and data streams has been presented. The service calls allow agents to communicate using an approach similar to the one used in object-oriented programming and simplify commonly-used protocols such as the request protocol. The approach is implemented by providing the calling agent with a proxy object representing the remote service of the called agent. When a service method is called, the call is automatically marshalled, transferred and executed by the remote agent. The remote agent, then, has the opportunity to either reject the service call through exceptions or execute it and provide the results to the called, which are once again marshalled and transferred to the calling agent.

The approach is asynchronous, allowing the agents to remain single-threaded and avoid potential problems with deadlocks and data integrity. While distributed systems have a certain fundamental complexity due to their inherent parallelism and asynchronous behavior, the extensive use of programming language constructs and close match with object-oriented features makes the approach fairly intuitive even for novice developers.

The service call as a basic building block is more complex than the messages used in message passing and potentially reducing some fine-grained aspects, hence it is still quite applicable for common problems and even more complex protocols can be assisted using specialized futures.

The presented stream implementation allows agents to stream binary data without consideration of detailed communication aspects. For this purpose two different

APIs have been described. The low-level API enables creation of virtual streams to other agents via the message service and the high-level API permits stream utilization as normal service parameters and return values. Besides, being easy to use the agent level, the conceived solution provides other advantages over typical TCP or other connections. First aspect is that failsafe connections and heterogeneous multihoming are supported by resorting to all available lower level transport means of the agent platform. Second, non-functional properties such as security settings can be safeguarded, and third, the approach can also be used in constrained scenarios in which no new connections can be opened.

Besides these aspects also performance of the streaming approach is an important factor for its usefulness in the practice. Using different example applications the performance had been compared with the original performance of a direct TCP connection and also a benchmarking had been performed. It revealed that the performance is close to a direct connection so that the comfort of using the APIs does not lead to a substantial trade-off decision between speed and usability.

As important part of future work we plan to add support for more non-functional aspects. In particular, we want to support stream priorities and unreliable streams suitable for audio and video transmission, where outstanding packets should be discarded and not resend.

REFERENCES

[1] Andersen, D.—Balakrishnan, H.—Kaashoek, F.—Morris, R.: Resilient Overlay Networks. Proceedings of the Eighteenth ACM Symposium on Operating Systems Principles (SOSP '01), ACM, New York, NY, USA, 2001, pp. 131–145.

[2] Bauer, B.—Müller, J. P.—Odell, J.: Agent UML: A Formalism for Specifying Multiagent Software Systems. International Journal of Software Engineering and Knowledge Engineering, Vol. 11, 2001, No. 3, pp. 207–230.

[3] Bellifemine, F.—Bergenti, F.—Caire, G.—Poggi, A.: JADE – A Java Agent Development Framework. Multi-Agent Programming: Languages, Platforms and Applications, Springer, 2005, pp. 125–147.

[4] Bellifemine, F.—Caire, G.—Greenwood, D.: Developing Multi-Agent Systems with JADE. John Wiley & Sons, 2007.

[5] Bellifemine, F.—Poggi, A.—Rimassa, G.: Developing Multi-Agent Systems with a Fipa-Compliant Agent Framework. Softw., Pract. Exper., Vol. 31, 2001, No. 2, pp. 103–128.

[6] Bless, R.—Mayer, C.—Hübsch, C.—Waldhorst, O.: SpoVNet: An Architecture for Easy Creation and Deployment of Service Overlays. River Publishers, Vol. 6, 2011, pp. 23–47.

[7] Braubach, L.—Pokahr, A.: Goal-Oriented Interaction Protocols. 5th German Conference on Multi-Agent System Technologies (MATES 2007), Springer, 2007.

[8] Braubach, L.—Pokahr, A.: Method Calls Not Considered Harmful for Agent Interactions. International Transactions on Systems Science and Applications (ITSSA), Vol. 1/2, 2011, No. 7, pp. 51–69.

[9] Braubach, L.—Pokahr, A.: Developing Distributed Systems with Active Components and Jadex. Scalable Computing: Practice and Experience, Vol. 13, 2012, No. 2, pp. 3–24.

[10] Braubach, L.—Pokahr, A.—Lamersdorf, W.: Jadex Active Components: A Unified Execution Infrastructure for Agents and Workflows. In Intelligent Hybrid Medical Complex Systems, Romanian Academy, 2012.

[11] Braubach, L.—Jander, K.—Pokahr, A.: High-Volume Data Streaming with Agents. In: Zavoral, F., Jung, J. J., Badica, C. (Eds.): IDC 2013, Studies in Computational Intelligence, Springer, Vol. 511, 2013, pp. 199–209.

[12] Bu, T.—Gao, L.—Towsley, D.: On Characterizing BGP Routing Table Growth. In IEEE Global Internet, Taipei, Taiwan, November 2002.

[13] Cabac, L.—Moldt, D.: Formal Semantics for AUML Agent Interaction Protocol Diagrams. In: Odell, J., Giorgini, P., Müller, J. (Eds.): Proceedings of the 5th International Workshop Agent-Oriented Software Engineering V (AOSE 2004), Springer, 2005, pp. 47–61.

[14] Cheong, C.—Winikoff, M.: Hermes: Designing Goal-Oriented Agent Interactions. In: Müller, J., Zambonelli, F. (Eds.): Proceedings of the 6th International Workshop on Agent-Oriented Software Engineering (AOSE 2005), Springer, 2005.

[15] Coulouris, G. F.—Dollimore, J.—Kindberg, T.: Distributed Systems. Addison-Wesley, 2005.

[16] Dinkloh, M.—Nimis, J.: A Tool for Integrated Design and Implementation of Conversations in Multiagent Systems. In: Dastani, M., Dix, J., El Fallah-Seghrouchni, A. (Eds.): Proceedings of the 1st International Workshop on Programming Multi-Agent Systems (PROMAS 2003), Springer, 2004, pp. 187–200.

[17] Ehrler, L.—Cranefield, S.: Executing Agent UML Diagrams. Proceedings of the Third International Joint Conference on Autonomous Agents and Multi-Agent Systems (AAMAS 2004), IEEE Computer Society, 2004, pp. 906–913.

[18] Foundation for Intelligent Physical Agents (FIPA). FIPA Abstract Architecture Specification, December 2002. Document No. FIPA00001.

[19] Foundation for Intelligent Physical Agents (FIPA). FIPA ACL Message Structure Specification, December 2002. Document No. FIPA00061.

[20] Foundation for Intelligent Physical Agents (FIPA). FIPA Request Interaction Protocol Specification, December 2002. Document No. FIPA00026.

[21] Foundation for Intelligent Physical Agents (FIPA). FIPA Request Subscribe Interaction Protocol Specification, December 2002. Document No. FIPA00035.

[22] Gosling, J.—Joy, B.—Steele, G.—Bracha, G.: The Java Language Specification, Second Edition. Addison-Wesley, 2000.

[23] Papazoglou, M. P.—Heuvel, W. J.: Service Oriented Architectures: Approaches, Technologies and Research Issues. The VLDB Journal, Vol. 16, 2007, No. 3, pp. 389–415.

[24] Postel, J.: Transmission Control Protocol. RFC 793 (Standard), September 1981. Updated by RFCs 1122, 3168.

[25] Rahm, E.—Do, H. H.: Data Cleaning: Problems and Current Approaches. IEEE Data Engineering Bulletin, Vol. 23, 2000.

[26] Ricci, A.—Viroli, M.—Omicini, A.: The A & A Programming Model and Technology for Developing Agent Environments in MAS. In: Dastani, M., El Fallah Seghrouchni, A., Ricci, A., Winikoff, M. (Eds.): Programming Multi-Agent Systems, 5th International Workshop (ProMAS 2007), Springer, Berlin, Heidelberg, 2007, pp. 89–106.

[27] Rooney, C.—Collier, R.—O'Hare, G.: Viper: A Visual Protocol Editor. In: De Nicola, R., Ferrari, G., Meredith, G. (Eds.): Proceedings of the 6th International Conference on Coordination Models and Languages (COORDINATION 2004), Springer, 2004, pp. 279–293.

[28] Sutter, H.—Larus, J.: Software and the Concurrency Revolution. ACM Queue, Vol. 3, 2005, No. 7, pp. 54–62.

[29] Thurlow, R.: RPC: Remote Procedure Call Protocol Specification Version 2. RFC 5531 (Standard), May 2009. Obsoletes RFC 1831.

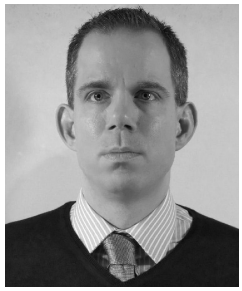
[30] Van Cutsem, T.—Mostinckx, S.—Boix, E. G.—Dedecker, J.—De Meuter, W.: Ambienttalk: Object-Oriented Event-Driven Programming in Mobile Ad Hoc Networks. Chilean Computer Science Society, International Conference, 2007, pp. 3–12.

[31] Whitestein Technologies. Semantic Communication User Manual, LS/TS Release 2.0.0 edition, 2006.

[32] Wooldridge, M.: An Introduction to Multiagent Systems. Wiley, Chichester, UK, 2nd edition, 2009.

[33] Xing, J.—Singh, M.: Formalization of Commitment-Based Agent Interaction. Proceedings of the 2001 ACM Symposium on Applied Computing, ACM, New York, NY, USA, 2001, pp. 115–120.

Kai Jander works as a research assistant in the Distributed Systems Group at the Computer Science Department of the University of Hamburg with a research focus on agile business processes in dynamic business environment. As part of this research he cooperated with Daimler AG in the DFG research project "Go4Flex" and contributed numerous workflow-centres and made other contribution to Jadex. Since 2013, he has provided his expertise in this area to help Uniique AG design and implement BI-focused workflows.



Lars Braubach currently works as a project leader in the Distributed Systems Group at the Computer Science Department at the University of Hamburg and in this role he initiated and conducted several national and international research projects. He received his Diploma in 2002, his Ph.D. degree in 2007 and habilitation in 2014 in computer science from the University of Hamburg. Since 2002, he published over 90 articles at international conferences, workshops and in journals. Starting in 2010, he and his colleagues began working with Uniique AG to create industry-grade highly-scalable and distributed BI software solutions based on Jadex.

Alexander Pokahr is a senior researcher in the Distributed Systems Group of the University of Hamburg. Since 2002 he has authored numerous publications in the area of software development for complex distributed systems. He is co-creator of the concept of "active components", a combination of components, services and agents, and co-developer of the open source active components framework Jadex. His recent interests include programming models for cloud computing applications.