High-Volume Data Streaming with Agents

Lars Braubach and Kai Jander and Alexander Pokahr

Abstract Agent technology is in principle well suited for realizing various kinds of distributed systems. Nonetheless, the specifics of agent technology render it difficult to implement data driven systems, even though these represent an important class of today's applications including e.g. audio or video streaming and file transfer. One central reason why agents are not especially helpful for these kinds of applications is the message-driven high-level interaction of agents. These kinds of interactions are meant to support high-level coordination of agents via speech act based actions, not transport of high volume binary data. Hence, currently agent developers need to resort to standard communication technologies like TCP or UDP to transport data, but this requires complex management of low-level aspects like connection handling at the application level and additionally might pose problems due to the requirement of opening additional ports. To better support also binary data transfer a new streaming approach is proposed, which introduces virtual connections between agents. 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 for failure tolerant transmission on basis of multiplexed data. The usefulness of the approach will be further explained with a real-word example application from the area of business intelligence workflows.

1 Introduction

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 dis-

Distributed Systems and Information Systems Group, University of Hamburg
{braubach, jander, pokahr}@informatik.uni-hamburg.de

advantages 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 this approach is that 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 order to motivate the requirements of a streaming approach we have analyzed an ongoing commercial project called DiMaProFi (Distributed Management of Processes and Files) in the area of business intelligence that is conducted together with the company Uniique AG.[1] The main objective of the project is to build a distributed workflow management system for ETL (extraction, transformation, loading) processes. These processes are in charge of moving and transforming huge amounts of data from different locations to a data warehouse, in which they are loaded to allow further inspections by domain experts. The workflow tool must allow specifying very different kinds of workflows as these are completely dependent on the concrete customer. From this scenario the following requirements were deduced:

- *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. In the project, the ETL processes are realized as agents. Here, it is often necessary to copy files to the executing workflow, e.g. if a subprocess is executed by a different node, data has to be transferred to it.
- *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. The infrastructures on which DiMaProFi is deployed depends on the customer, ranging from banking scenarios with strong restrictions to internet providers with fewer security policies but distributed networks. Constraints can therefore vary to a high degree.
- *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. Furthermore, DiMaProFi deals with big data. This means that it is crucial to avoid complete retransmissions of large files if parts already have been successfully transferred.
- *Non-functional properties:* The quality of service characteristics, i.e. non-functional properties, of the transfer should be configurable. Important properties include e.g. security, reliability, and priorities. In the workflow scenario, customers often execute different kinds of ETL processes at the same time. As these processes have different deadlines, it is important to

[1] `http://www.uniique.de/`

allocate execution resources according to these deadlines. Part of these resources are non-functional properties of data transfers.

In 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. The next Section 2 presents the approach itself and its implementation. Thereafter, Section illustrates 3 the usage of the approach with the ETL workflow application. Section 4 compares the proposed approach to existing solutions and Section 5 gives some concluding remarks and a short outlook to planned future work.

2 Data Streaming Approach

The requirements of the last section have been carefully analyzed and strongly influenced the design of the streaming architecture presented later. Here the findings are shorty summarized according to the already introduced categories. As an additional point the agent integration has been added as the characteristics of agents also determine the set of possible solutions.

- *Location transparent addressing*: This implies that a connection should have an agent as 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 to 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 being an option in addition to the traditional message sending approach.

2.1 General Architecture

In Fig. 1 the streaming architecture is depicted. 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 [6] in the sense that it is assumed that on each agent platform a MTP (message

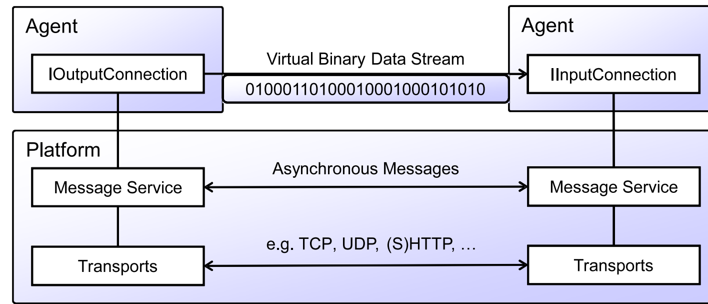

Fig. 1 Stream architecture

transport service) exists that is capable of sending asynchronous messages to agents of the same and other platforms. For this purpose it makes use of different transports, which utilize existing communication technologies such as TCP, UDP or HTTP to transfer the data.

2.2 Stream Usage

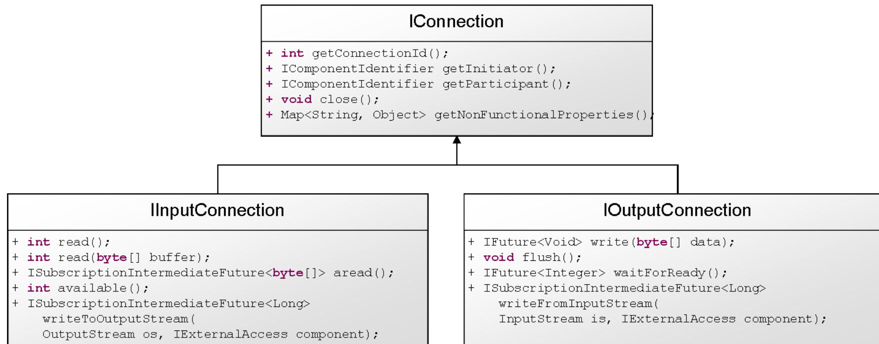

Fig. 2 Stream Interfaces

In order to better understand the envisioned usage from an agent perspective, in Fig. 2 the important streaming interfaces are shown. Each connection (IConnection) has a connection id as well as two endpoints, an initiator (agent) as well as 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.

An output connection (IOutputConnection) is used to write binary data in chunks to the stream (write). 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 for automatically processing a 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 for automatically writing all incoming data to a Java output stream (writeToOutputStream).[2]

2.3 Low-level API

```
01: IFuture<IOutputConnection> createOutputConnection(IComponentIdentifier initiator,
02:         IComponentIdentifier participant, Map<String, Object> nonfunc);
03:
04: IFuture<IInputConnection> createInputConnection(IComponentIdentifier initiator,
05:         IComponentIdentifier participant, Map<String, Object> nonfunc);
```

Fig. 3 Extended message service interface

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 Fig. 3. 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 result of the call the corresponding connection instance is returned.

An agent that is used as 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 parameter to the streamArrived method call. Having received such a callback the receiving agent is free to use as it deems appropriate. Of course, it can also do nothing and ignore such incoming stream connection attempts.

[2] Please note that in contrast to Java streams all connection interfaces are non-blocking although the method signatures look similar. 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 during it waits for the blocked call to return. In the interfaces different future types are used to render them asynchronous. A future represents a placeholder object that is synchronously returned to the caller. The real result will be made available to the caller once the callee has finished processing and set the result in the future [7].

2.4 High-level API

For active components [5], 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 for 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, which allow for fetching the corresponding opposite connection endpoint.

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.

2.5 Implementation Aspects

The proposed concept has been implemented as part of the Jadex platform [5]. The implementation distinguishes different responsibilities via different layers (cf. Fig. 1). 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 by a virtual stream control protocol, which is based on well-established TCP concepts.[3]

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. The output connection keeps track of lost packages and resends them if necessary. Furthermore, the connection realizes flow control by using a sliding window that adapts the sender's connection speed to the speed of the receiver. Connection set-up and teardown is handled via specific handshake messages. 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

[3] 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.

protocol. Whenever the API is used to create a virtual connection (cf. Fig. 3) 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 have to support streaming.

3 Case Study

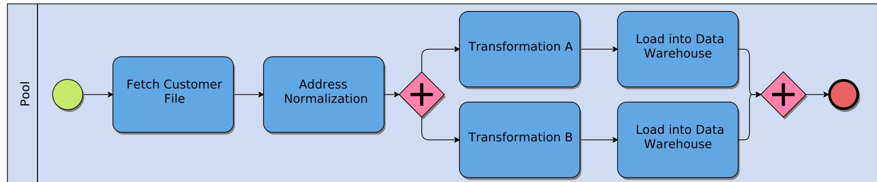

Fig. 4 An ETL process loading a file, transforming and writing it to the data warehouse

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.

Fig. 4 shows an example for such a process. When a customer file is stored on the file server, the monitoring process is notified and initiates the ETL process on a remote machine. The process requests the binary stream for the file server (fetch customer file) and stores the file 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. 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.

The code of the fetch customer file task, which uses the high-level streaming API, is depicted in Fig. 5. It consists of the (reduced) interface of the file service, which offers a method fetchFile() to retrieve a remote file.[4] As parameter the method takes the local file name and as 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 the input connection is obtained by

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

```
// File service service method
01: public IFuture<IInputConnection> fetchFile(String filename);

// Fetch file task except
01: IInputConnection icon = fileservice.fetchFile().get();
02: FileOutputStream fos = new FileOutputConnection(tmpfolder+"/"+filename);
03: icon.writeToOutputStream(fos, agent).get();
04: fos.close();
```

Fig. 5 Code excerpts of fetching a remote file

calling the fetchFile() service method of the file server (line 1). Afterwards a file output stream for the temporary file is created (line 2) and the whole file content is automatically written to this file output stream by calling writeToOutputStream() (line 3). 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 all data has been written to the file. Finally, the stream is closed.

4 Related Work

	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	-	-	+	-

Fig. 6 Streaming support requirements and support by different approaches

As mentioned in Section 2, 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 agent 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 Fig. 6.

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 [2], 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 [3]. However, this forgoes the advan-

tage 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) [4]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 streaming support.

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

5 Conclusions and Outlook

In this paper a concept and implementation has been presented that 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. In the following, it will be summarized how the proposal helps achieving the initial requirements.

- *Location transparent addressing*: The architecture makes use of the existing agent based addressing, i.e. each stream has an initiator and particiapnt agent identifier. An agent sending data to another agent can use the target's agent identifier to create a virtual stream connecting both without knowledge where this agent resides.

- *Infrastructure traversal*: The layered model allows for a clear separation of concerns and enables the streaming mechanism to utilize the existing FIPA transport scheme. As a result, if platforms manage to reach one another through some channel, it can be used simultaneously for standard messaging as well as for binary streaming. Stream data is automatically fragmented into small packets, which can be multiplexed with other data.
- *Failsafe connections and heterogeneous multihoming*: The message service uses multihoming by setting up different transports. Connections are virtual, using all transports available, i.e. the message service will try to send messages (binary as well as standard) via different transports until it fails and no alternatives are available.
- *Non-functional properties:* Streams can be initialized with non-functional properties. These are used by the connection and message service to handle the connections properly. In the current implementation only non-functional criteria for security related aspects are supported.
- *Agent integration*: Streaming is available at the agent level by streaming APIs. These completely relieve a developer from low-level communication issues. The integration supports the reactive nature of agents by using a non-blocking approach without additional threading within an agent.

Besides these aspects also performance of the streaming approach is an important factor for its usefulness in practice. Using different example applications the performance has been compared with the original performace of a direct TCP connection. The testing has revealed that the performace is very close to a direct connection so that the comfort of using the APIs does not lead to a 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. D. Andersen, H. Balakrishnan, F. Kaashoek, and R. Morris. Resilient overlay networks. In *Proceedings of the eighteenth ACM symposium on Operating systems principles*, SOSP '01, pages 131–145, New York, NY, USA, 2001. ACM.
2. F. Bellifemine, F. Bergenti, G. Caire, and A. Poggi. JADE - A Java Agent Development Framework. In *Multi-Agent Programming: Languages, Platforms and Applications*, pages 125–147. Springer, 2005.
3. F. Bellifemine, A. Poggi, and G. Rimassa. Developing multi-agent systems with a fipa-compliant agent framework. *Softw., Pract. Exper.*, 31(2):103–128, 2001.
4. R. Bless, C. Mayer, C. Hübsch, and O. Waldhorst. *SpoVNet: An Architecture for Easy Creation and Deployment of Service Overlays*, pages 23–47. River Publishers, 6 2011.
5. L. Braubach and A. Pokahr. Developing Distributed Systems with Active Components and Jadex. *Scalable Computing: Practice and Experience*, 13(2):3–24, 2012.
6. Foundation for Intelligent Physical Agents (FIPA). *FIPA Abstract Architecture Specification*, December 2002. Document no. FIPA00001.
7. H. Sutter and J. Larus. Software and the concurrency revolution. *ACM Queue*, 3(7):54–62, 2005.