Annales Univ. Sci. Budapest., Sect. Comp. 31 (2009) 3-22

An Agent Based Architecture Style for Application Integration

Jari Peltonen and Mikko Vartiala

(Tampere, Finland)

Abstract. In order to realize the time-to-market needs of an information system, it is essential that integration of existing applications is reasonably utilized. In addition, also flexibility, maintainability, and incremental development are typical requirements for software architectures, led from the typical business and marketing needs. To succeed in fulfilling these requirements, attention must be paid to various architectural viewpoints, including the location and granularity of business logic, as well as simplicity and loose coupling in the architecture. In this paper, the main focus is in integration of applications in a maintainable and flexible way that supports incremental development. We consider our agent based software architecture style, and show how higher level of abstraction in dependencies, as well as relocation of the strategic business logic to agents, provides a feasible solution to the above-mentioned needs. To validate the approach, we have implemented a framework for agents and used it to create the architecture for a process support environment.

1. Introduction

The constant business goals in software development are to use less time and money for development and maintenance, get the products faster on the market, and still preserve appropriate quality and the ability to adapt to the changing world. From the technical point of view, this indicates that high level of flexibility and maintainability is typically required, and incremental development and integration of existing applications are often preferable ways of making software.

Maintainability is important simply because considerable amount of time and money is used to the maintenance of a system, especially in systems where the life span of the system is long. *Flexibility* is necessary for adaptation to, e.g. technological trends and an increasing variety in device and communication environments. *Incremental development* and *integration* of existing applications may considerably help to fulfill quality, cost, and time-to-market requirements set.

Especially in the information system development, integration of existing systems, components, and applications can generate great savings and raise the quality of the system. The need for integration includes also possible legacy systems and commercial off-the-shelf products. Furthermore, integration of legacy systems is often more of a necessity than a choice. Incremental development process is a good choice due to the fact that the time-to-market requirement is typically not so much about the time when the full system is functional, but more about when some functional system is on the market.

Incremental development, flexibility, maintainability, and integration are best utilized when they are considered already in the architecture of the system. To succeed in fulfilling these requirements, simplicity of the architecture and loose coupling of the architectural entities are proven to be successful paradigms. For instance, service orientation [18] is largely based on these ideals.

However, the difficulty of managing dependencies in any architecture depends also on how the functionality and data are divided across the architecture. Specifically, the location and granularity of functionality are traditional issues in the design of an information system. Too often, even a simple functionality concerning a single aspect in a system is wide spread across the architecture. This is not just counterproductive, but this kind of scattering of functionality efficiently prevents all attempts, e.g. for easy maintenance and incremental development. In addition, badly designed architecture and communication within it easily leads to transmitting excess data that can affect negatively, e.g. on the performance and resource requirements of the system.

The main focus in this paper is in integration of applications in a maintainable and flexible way that supports incremental development. We gain this by providing a simple infrastructure model, promoting loose coupling by higher level of abstraction in dependencies, and locating each strategic business logic case to a single place – an agent. The architecture style also aims at, e.g. optimizing communication within the system.

The current trend of using messages in application integration is considered in Section 2. In Section 3 an overview of our agent based architecture style is given. In Section 4 the implementation of our framework for agents is presented, and Section 5 shows how we have used the framework to create the architecture for a process support environment. Section 6 summarizes and discusses the approach and Section 7 discusses some related work. Finally, in Section 8 we give some concluding remarks.

2. The current trend in application integration – Messages

2.1. Why messages?

Messages are often seen as the most versatile option for application integration over file transfer, shared database, and remote procedure calls (RPC) (e.g. [9]). File transfer and shared database approaches are solutions for sharing data, but not functionality. RPC again makes it possible to share functionality, but couples the applications tightly to each other at the same time. In addition, remote procedure calls are slower and much more likely to fail than local ones, and due to the synchronous nature of communication, a failure in one application may break down the whole system. File transfer, as an integration approach, is asynchronous and decouples applications well, but does not transmit the data in real time.

Messaging aims at mixing the good attributes of file sharing and RPC by allowing near to real time data transmission and functionality invocation asynchronously. Asynchronous communication is one of the key points when aiming at loose coupling among applications. Sending a message does not require all participating systems to be available at the same time, and the sender does not have to wait the response, but it can continue on doing other things. In addition, any procedure calls a message actuates are local, which makes the system more reliable.

Architectural styles like Service Oriented Architectures (SOA) [18] and Enterprise Service Bus (ESB) [5, 10] emphasize loose coupling by relying on indirect asynchronous message based communication. They work conceptually on higher level than, e.g. traditional client-server architectures, since they do not discuss physical clients or servers, but logical services and their consumers. This detaches the architectures from physical world, and thus from physical addresses. The service consumers also tell what services they want, not how they will be performed. Higher level of abstraction in dependencies is a favorable solution in application integration since it makes loose coupling as the central pattern in the architecture.

2.2. Deficiencies of message based systems

In a message based system, a close to real time communication is achieved by sending a lot of small messages and letting the receiver to know immediately when a message is available. This generates easily a great amount of network traffic, which may become a problem in larger and more complex systems. In addition, not all of the messages are small and simple, since they are used to transmit all the information and related meta-data in the system. Hence, messaging may put a heavy burden on a communication channel. This is a problem in any environment, but especially in the ones where the communication channels are thin (like mobile environments).

Due to various schemas and data formats in different applications, each message goes trough a transformation chain, where the message is first formulated, translated to a common format and sent, and in the other end it is received, parsed, interpreted and actuated. This requires some processing power, as well as causes lag for the communication. In addition to the minor inconveniences caused by latencies, the total completion time may grow considerably.

Since the message must be interpreted in the receiver end, both the sender and receiver must understand the exact semantics of the message. This means that a single concern in functionality is always divided across the architecture, and the comprehension, maintenance, and testing of such concern gets very hard. The problem is even worse when the needed functionality is complex, and there is a need for several messages to get a single thing completed.

Due to need to minimize the network traffic and to simplify the communication, a high granularity in services would be favorable. However, e.g. maintainability and re-use of services would benefit from lower granularity. This is not solely a challenge of message based systems, but more a balancing issue of any system where there is a need to communicate over network. A typical solution is to compose higher level “services” of lower level components, or to introduce some middleware. In the case of the service oriented architectures, the only lower level components are other services, which sets the lower limit for rational granularity rather high. In addition, the basic problems of message based systems do not vanish as long as the only way of communication is by using messages.

Basically, any sequence of service requests in a message is a sequence of commands and can hence be considered as a script. The language for specifying a script just does not have the power of typical scripting languages. There are no other ways in messages to react dynamically for varying or exceptional situations either. Not very much can be done, for example, if a service fails during the execution. The service may be able to send an error message to the service consumer, but again, an amount of messages are sent to various places. In addition, there must be some code to react to that kind of messages too – in all the service consumers who might be interested.

As an example, let us consider a situation where a service consumer wants to calculate a trend based on a large amount of information on several services. This means that there are several related messages either sent one by one to the services and then the results are collected and interpreted in the consumer, or there is a chain of messages where the information from previous service is forwarded to next one, and the following service again interprets the data it gets.

Particularly, if the data in services depend on each other in the calculation, or the way of performing the calculation is dynamical (e.g. depending on the consumer or data provided by the services), there is either a huge amount of network traffic, or the services become unnecessary complex. Either way, the functionality needed for performing a single calculation is spread across the architecture, the business sequence gets hard to comprehend, maintain, and test, and it is hard to get the whole system robust and fault tolerant.

3. An agent based architecture style

3.1. An overview of the approach

The good ideas of promoting loose coupling by higher level of abstraction in dependencies, as well as the simplicity of architecture style are worth preserving. However, to overcome the deficiencies of message based communication mentioned in Section 2.2, we wish to promote locating each strategic business logic case to a single place – an agent. We believe this to be a more feasible solution for application integration than messaging.

The general idea of the agent based architecture style is that there is an infrastructure offering services for agents, which use the infrastructure to move around and to achieve their goals. It is notable that typical agents are not very complex; on the contrary, most often they are simple task based agents with a predefined behavior. Additionally, one agent should only be related to a single task for simplicity.

To make a clear distinction between the entities on different abstraction levels, we present the approach in three meta-levels, where a higher level architecture defines the possible instances of lower level architectures. As seen in the vertical axis in Fig 1 the levels are from the most abstract to the most concrete: meta-architecture, system architecture and runtime architecture. The meta-architecture, i.e. the architecture meta-model, describes the entities that can be used to define new system architectures. Basically, a meta-architecture

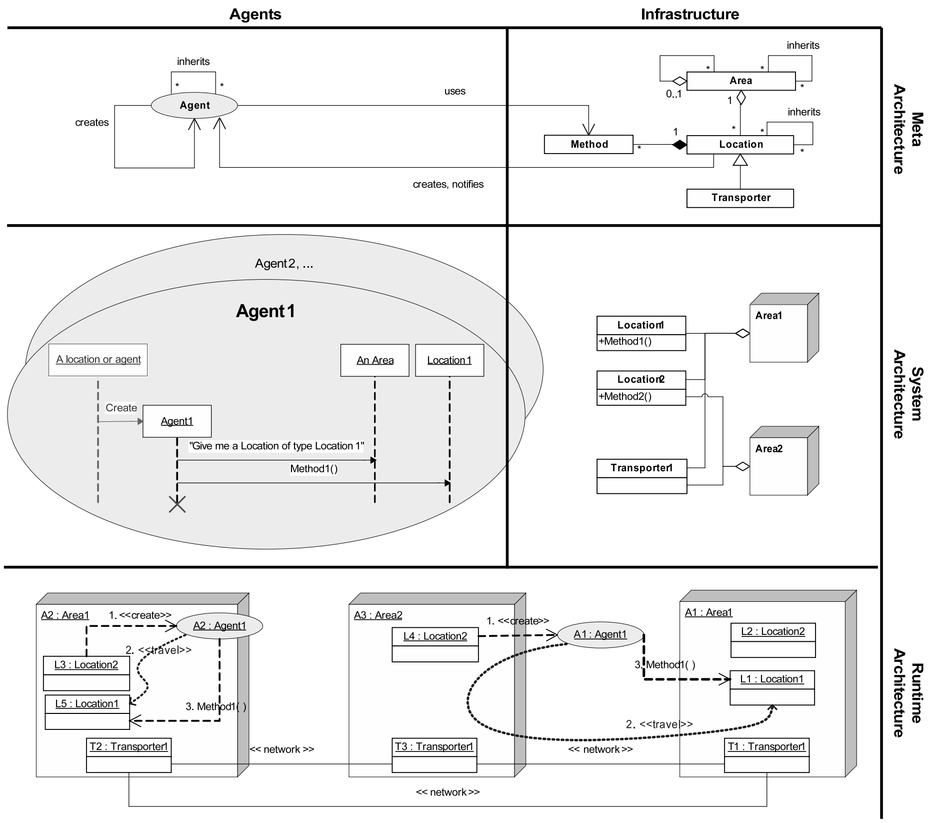

Figure 1. The three metalevels describing the agent based architecture model

is an architectural style defining a language for specifying possible architectures according to that style. In this sense, all the architecture definition languages can be seen as architectural styles.

System architecture is the logical architecture definition of a concrete system and runtime architecture is a possible, physical, runtime instantiation of the system architecture. There is also fourth level, meta-meta level, which defines a language for specifying meta-architectures. In this case we use OMG Meta Object Facility (MOF) as such language [17]. Besides that the architecture is divided vertically to meta-levels, it is also divided horizontally to infrastructure and agents as seen in Fig 1. That is, we separate the business logic from the underlying infrastructure.

The meta-architecture of the infrastructure, as shown in the upper right corner of Fig 1, consists of areas, locations, methods of locations and transporters. An area represents one group of locations typically located in one computer.

Locations offer different kinds of services to agents through their methods and they can also create new agents when something needs to be done. Typical locations include user interfaces, as well as interfaces to databases and various other applications.

Transporters are special kind of locations connected to each other. They are used for transporting agents to remote areas. The architecture style allows three different forms of traveling: Agent tells the infrastructure 1) only the type of the location, 2) the type of the location and the type of the area or 3) the type of the location and the ID of the area. The locations, areas, etc. are meant to be built in a way that they do not know anything about the functionality provided by other entities in the infrastructure.

The agents, seen on the left side in Fig 1, use the functionality offered by the infrastructure to achieve their predefined tasks. More specifically, the agents move among different locations, possibly located in different areas, and use the methods of the locations to achieve tasks. The agents do not need to know anything about the runtime architecture, but they can rely on their knowledge of the description of the system architecture. More specifically, they typically only need to know directly the types of the locations they want to use. The only things that get transferred between areas are agents.

The architecture does not limit the amount or type of the above-mentioned entities in any way. On the contrary, one of the key points is that it should be made as easy as possible to expand any system using this architecture by adding new agents, locations, areas and transporters to it. This helps to achieve the needed flexibility, customizability, and incremental development requirements. For the same reason, the maintenance of the system is straightforward.

3.2. System and runtime architectures

System architecture is the description of the architecture of a concrete system. It is achieved by instantiating the meta-architecture in any way the architect desires. A possible example of system architecture can be seen in the middle part of the Fig 1. The example consists of two agents, two areas, two locations and a transporter, named according to their types. Notable in the example is that both areas have Transporter1 and Location2, but Area1 has additionally Location1. A reason for this might be that Location1 requires some special resource or processing power not available in a normal workstation, thus a more efficient server is required to run Area1.

What cannot be seen from the figure is what kinds of connections are allowed by Transporter1. Generally, the type and number of possible connections depends entirely on what kind of transporters there are in an area. For example, Transporter1 could allow connecting to an unrestricted number of other transporters, or it could only allow one connection to a transporter of type Transporter1. In this case there can be an unrestricted number of connections.

All the program code, including the behavior of the agents, is defined in the system architecture level. That is, agents rely typically only on the logical architecture elements instead of physical ones. As an example, the behavior of a simple agent type (Agent1) is illustrated in Fig 1 with a sequence diagram like presentation. A location or agent creates an agent of type Agent1 whenever they need a service provided by such an agent. When an agent is created, it typically gets parameters that guide its execution, as well as other information, like the ID of the area where it was born. The example agent uses Method1 in a location of type Location1 to perform its task. Since the programmer of an agent cannot typically know whether there is a needed type of location nearby, the needs must be indicated to the infrastructure (e.g. the area where the agent currently is). After the infrastructure moves the agent to an appropriate area and location, the wanted method can be used.

Runtime architecture consists of all entities and their states of a system in one moment during runtime. It is possible to have an unlimited number of different runtime architectures using the same system architecture, because typically the amount of entities is not constrained in any way. An example of a possible runtime structure is seen in the bottom level of Fig 1. This runtime structure consists of three areas, and as defined in the system architecture, each area has an instance of Transporter1 and either one or two locations. All of the transporters are connected to each other over the network, and hence they form a kind of a peer-to-peer network in this case. The situation in the example, three areas and two agents, is not caused by any restrictions; an equally possible case would a runtime situation with, say, tens of areas and hundreds agents.

The dashed lines in the bottom level of Fig 1 show the runtime behavior of two different instances of Agent1. Both of them are created by a location of type Location2. The leftmost dashed lines show what happens when such an agent is invoked in Area1. When the agent comes to a situation where it needs to use Method1, it indicates to the infrastructure that it needs to use a location of type Location1. Since a location of that type is located in the same area, the agent is moved there. After the short travel the agent calls Method1 and decides that it has done everything it needed and thus the agent stops there.

The rightmost dashed lines show the behavior of Agent1 when it is created in an area of type Area2. As a distinction from the previous example, there is no Location1 in the area where the agent is created. Thus, when the agent wants to use Method1 of Location1, the infrastructure transports it to an area which has a location of type Location1, in this case the area A1 is chosen. The second line is a composition of all the events that occur during that travel. After the traveling the agent uses Method1 of the location L1 and stops.

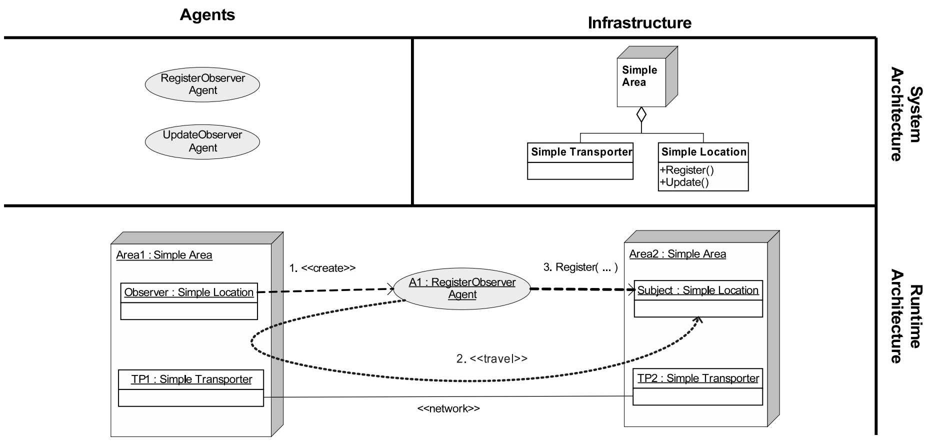

Figure 2. Using observer-pattern in agent-based architecture

3.3. An example: observer-pattern

The simplest complete system architecture to support observer pattern [4] can be created with five entities in the system level as seen in Fig 2. The meta-level is not described anymore as it is same for all system architectures. In the infrastructure side there is one area, Simple Area, which consists of two locations, Simple Transporter and Simple Location. Simple Location works as both the observer and the subject, and it offers methods Register and Update. To achieve the functionality needed in the pattern we need two agents. RegisterObserver-agent registers an observer to a subject and UpdateObserver-agent is then used to update the registered observer.

In the bottom level of Fig 2 there is the runtime architecture with two instances of Simple Area. The Simple Location in the leftmost area works as an observer and the Simple Location in the rightmost area works as a subject. The dashed lines in Fig 2 show the sequence of events during the lifetime of a RegisterObserver-agent. The sequence starts when the leftmost Simple Location wants to register itself to the Subject and creates an agent for this purpose. The needed parameters are also given to the agent at this point. These parameters include at least the type of the subject-location and the ID of Area2, because the agent needs to know exactly who to register and to whom. Additionally the initialization data could include, e.g. the type of events that the observer is interested in. The second line is a composition of all the events that occur during the travel from the observer to the subject. Line 3 shows the actual registration of the Observer-location. After that the agent stops and is destroyed.

4. Implementation of a prototype framework

4.1. The infrastructure supported by the framework

To validate the approach, we have implemented a prototype framework. The framework implements all described entities in the meta-level (location, area, agent, and transporter) of the architecture and makes it possible to specialize system level architectures from it. The framework also implements several other helpful entities to make the implementation of a working system easier. There are also some implementation specific details not part of the architecture model itself. These details are described in the following paragraphs.

All locations in the infrastructure offer some basic functionality to agents. They allow the agents to travel to other locations and to redirect an agent to a transporter if the wanted location is in another area. They also allow asking the current area and the type of the current location. The type of the location is important information, since the agents typically navigate in the infrastructure using them. Areas only know the types of their locations and have no other knowledge of them or other areas, i.e. areas are autonomous and running an area does not directly require the presence of any other areas. All areas have a type and an ID; these can also be used by agents to move among them. Each area also has at least one transporter.

Agents are transported by first serializing the state and data of an agent in a transported, then creating a similar agent at another transporter in a remote location and deserializing the state and data for this new agent. One transporter can have multiple connections to other transporters. Common functionality to all transporters is that they can be asked for all currently connected areas and to transport an agent to any of them. Common to the whole framework is that it must take care of concurrency, network communication and all other things that are not related to the business logic, so that the agents can focus on implementing the non-quality requirements of the system.

4.2. The general characteristic of agents in the framework

An agent has current location, current state and a home area. The home area tells where an agent originates from, and where it should navigate if it wishes to come back from a remote location. Current state is used to determine what the agent has done, and what it should do next. There is no predefined state behavior or other constraints for the states of the agents, but it is hard coded to them, i.e. it is left to the creator of an agent to use the agents any way she prefers.

Agents can create other agents and in some cases even interact with them, but they can only coordinate their movement according to locations and have no knowledge of other running agents unless a location provides this information. Agents cannot create themselves, but otherwise their lifespan is completely handled by themselves. Agents can duplicate themselves at will and they are never destroyed unless first requested by themselves.

Agents do not directly need to handle lower level things, like concurrency, in any way. The framework takes care of those. Of course there can be many agents under execution at the same time, but agents should not have to care about this. Still, they might have to wait before the execution of any called method of any location. The order of the queued agents may also change in some cases. Therefore, it is not always guaranteed that a preceding agent can use a location before a later arrived agent. It is also possible that when calling two non-related methods in the same location another agent comes and calls the same location in between the two calls.

The whole execution path of an agent should typically not be considered as a transaction since the framework does not currently offer any means to recover an agent which is in a disconnected area or to detect the loss of an agent. Agents can of course try to offer quality of service, but it is usually easier to just try to notify the user about an error and then leave the rest to her.

5. An example system - a process support tool

5.1. An overview of the example system

We have used the agent based architecture style to implement a process support environment. The environment is used to execute a software development process with several tools and developers. The process support environment must make it possible to define the used process and the users must be able to see the state of the process and control it. The state of the process, as well as the artefacts produced and used need to be persistent. Because of several developers, the process needs to be synchronized among all of them. It is also essential that existing tools, used by the developers, can be integrated to the environment. The inherent nature of software development is such that the process, tools and environment may change for every project. Hence, the abovementioned definitions, integrations, etc. must be very flexible. Additionally, for performance, usability, etc. reasons, it must be possible to execute process activities and use tools both on local and remote computers.

We have made several architectural decisions regarding the environment. Only the most important ones are listed here. The process is defined as a Visiome script [19], which is run in a Visiome Engine. On top of Visiome Engine runs a model processing platform called xUMLi [1, 20]. Both of them will be part of the architecture and existing modeling tools (like Rational Rose) are integrated through xUMLi. The existing tools could of course be also integrated directly to the architecture, but since there is an existing implementation fulfilling our needs, we do not need to do that. A frontend is needed for following and controlling the state of the process. It was decided that the persistency is handled by saving the state of the process to a database and the artefacts to a version control system. A process backend is used to make it simple to synchronize the process among different frontends and to allow remote processing at the backend.

5.2. The system architecture

The prototype framework was used to implement the example system, i.e. we have inherited all the used locations, areas, transporters and agents from their corresponding base classes. These inherited entities can be seen in the upper part of Fig 3. The current instantiation of the architecture can basically be seen as a kind of client-server architecture. The system infrastructure consists of two different kinds of areas, Backend and Frontend. Both of these have their own transporter. The frontend transporter can only connect to one backend transporter at a time, and the backend transporter can only receive connections from frontend transporters. To add support for multiple backend areas either the backend transporter should offer functionality to connect to a backend transporter or a frontend transporter would have to be added to the backend area.

Common locations for both of the areas are Database and VersionController. Database-location offers an interface to the shared database of the system and VersionController-location offers an interface to use the shared version control of the system. FrontendEngine and UI are only located at the Frontend. FrontendEngine handles the communication to a Visiome Engine at the frontend and UI is a single user interface. ProjectHandler is only located at the Backend. It manages the relations between users and projects and creates new visiome engines. There are several different kinds of agents in the architecture; these include a StartProject-agent and an ExecuteActivity-agent. Most of the agents in the architecture are typically started by a software developer who uses the UI in a frontend.

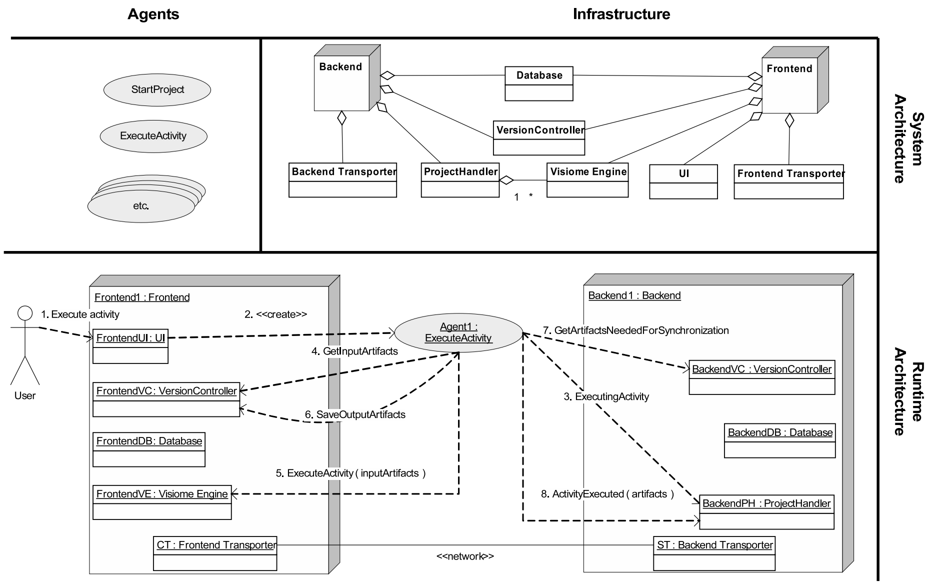

Figure 3. The specialized architecture and a sequence of an ExecuteActivity-agent

5.3. An example run-time architecture

There is an example runtime architecture with one Frontend area and one Backend area in the bottom of the Fig 3. The dashed lines show the sequence of events during the lifetime of a bit more complicated agent, an ExecuteActivity-agent. This agent executes an activity at the area in which it was created. The traveling between the areas and locations has been omitted for simplicity.

The sequence starts when a user implies her wish to execute an activity. Then an agent is created, and it uses the locations in the order shown by the numbers. First it must travel to the backend and use ProjectHandler to lock the activity so that no other user can execute it at the same time. At this point the agent fetches the needed input files of the activity from the VersionController and starts to execute the activity at the Visiome Engine of the frontend. The activity itself can be of several different types, including an automatic activity with no user intervention or an interactive activity which requires interaction during the execution. After the execution the output is saved to VersionController at the frontend and synchronization is done at the backend using the outputs.

5.4. Experiences

The framework and the complete system were implemented quite painlessly and successfully in reasonable time; therefore the case study can be considered a success. Also, if only the user interface is left out of the row count of the code, then the implemented system architecture has only a little more code lines than the implemented meta-architecture, i.e. the framework.

The division to the framework and to the system itself was quite viable and the framework implements several functionalities in their entirety. These include the transporting of the agents; including the moving over network; handling of the concurrency and general structures for managing locations and agents. Additionally there was only a minimal need to put non-requirements related things in the implementation of the system architecture. In the example system the methods of the locations are individual in the sense that there is no session between locations and agents using them, i.e. the locations do not provide methods which require that a specific agent calls them one after the other.

6. Discussion

The presented architecture style is relatively simple, but it still considers the requirements set in Introduction Section. Any architecture made according to the style is loosely coupled, since the system architecture level entities have no direct dependencies and the agents are only dependent on the types of locations instead of any specific locations. In other words, the agents have information about the logical system architecture, but not about the physical runtime architecture. Additionally the locations used by agents merely offer methods to them; they do not have to know anything about the way they are used, for example, interpret messages. The agents are the only entities having knowledge about their behavior.

Typically, agents are directly related to a single business sequence, i.e. each agent implements an aspect, like the whole higher level communication over different components during the order making sequence. This provides easy viewing of the completeness of an aspect, and as important consequences, easy maintenance of them and possibility to add new aspects to the system incrementally.

Incremental development is also achieved by making it easy to add new areas, locations, transporters, and agents. By using inheritance, it is easy to create extended versions of existing entities. In addition, agents can be “composed” of other agents, i.e. they can use other existing agents to perform their tasks. For example, to add the functionality of sending a message to all currently connected testers in a project only a new simple agent would have to be created. If there would already exist a SendMessage agent, the new agent could use it, or be specialized from it. Adding a new type of an agent does not change anything else in the system.

Agents can also easily react to varying or exceptional situations, since from the viewpoint of an aspect, all essential things happen in a single agent. An agent can, for example, spontaneously fetch more information if needed, or dynamically decide the area, i.e. computer, in which the processing is done, e.g. in complex calculations. For instance, in the trend calculating example presented in Section 2, an agent could travel through all of the information providers sequentially and make decisions regarding the rest of the calculation on-the-spot, thus reducing the amount of network traffic, since some information could be discarded immediately. More generally, agents aim always at optimizing network traffic by their very nature, i.e. they only carry with them the currently needed information.

In a way the agents can be compared to messages in a message based system, the difference being that a message implicitly tells the system what to do, but an agent explicitly has the information in it. Additionally, an agent has the whole expression power of a decided programming language at use, and they are always executed locally if considered from the viewpoint of a location. No transformation chains or extra data formats are needed as the agent designers generally understand the interfaces of the locations. Whether the interfaces of service providers, that is locations, are descriptive or procedural does not matter. In any case they are close to the service itself and thus no transformation is needed.

The communication model of the instantiated architecture can be anything the developer wishes, it completely depends on what kind of transporters there are in the system. For example, a peer-to-peer network is achieved by having a peer transporter, which can connect and simultaneously receive multiple connections from other peer-transporters. The agents do not care about the runtime architecture, as long as they can move among locations they need. Many quality requirements not explicitly defined in the architecture style, including reliability, security, QoS, etc. could be handled by adding specific functionality to transporters. They could, e.g. encode the agent states after serializing etc.

In our approach the granularity problem is divided into two parts, there is the granularity of the methods of the locations, and the granularity of the agents. Typically the methods are more fine-grained and the agents more coarse, as the agents use the methods to aggregate higher-level functionality. This way a designer in need of some functionality can first browse the existing agents, and if no proper agent exists, she can create a new one to use, not only the methods of the locations, but also other agents.

The implementations of the agents are typically fairly simple, consisting mostly of straightforward snippets like 1. To infrastructure: Transport me to LocationX, 2. Call method XY of that location, 3. Change internal state of the agent according to the return value of method XY. Agents in the example system described in Section 5 were usually about 50-100 lines of code, the longest being about 250 lines. This could have been made even smaller by some more careful planning and additional optimization of the framework and perhaps by using for example, Python to implement the agents.

7. Related Work

SOA [18] has been successfully used at several integration projects, including [24] and [25]. To integrate a system using the agent based architecture with an external SOA system is, at least in theory, relatively easy. It could, for example, be done by creating a location which accepts external SOA-messages and converts them to the right agents. Also the same location could convert method calls made by agents to SOA-messages and send them to the right service providers.

There exists a lot of research done in a multitude of areas involving agents directly or indirectly. For instance, [14] gives an overview of agent concepts and applications of agent technology. Baumann et al. [2], Lange and Oshima [12], and Gray et al. [8] have found similar benefits of using agents as we pointed out. The experiences with first- and second-year undergraduates successfully developing D'Agent applications [8] also suggested that agents are easier to understand than message- or RPC-based techniques.

There are also numerous agent-based architectures, infrastructures and middlewares, including Mole [2], the Aglet API [11], Open Agent Architecture (OAA) [15], D'Agents [8], RETSINA [23] and Hermes [7]. The middleware presented in Hermes has been successfully used to design an agent-based tool integration system [6]. A summary of several projects using agent technology for enterprise integration and supply chain management is presented in [22]. Existing agent architectures are discussed and an architectural model for mobile agent systems is described in [21]. Additionally, [16] considers the use of agents in electronic business, including complex integration of existing infrastructures.

A common difference with our approach and many of the mobile agent systems is that our focus lies in simplicity which is achieved by restricting the mutual communication of agents to be between agents and locations. This allows us to support flexibility in a controlled manner while still keeping the system easily maintainable. A more specific difference with other agent-based architectures is that we have a special entity called as location, which provides local services. Our service provider is called Location, instead of service agent or static agent, because of the fundamental differences between agents and locations in our system. The most relevant differences being that locations are not mobile or goal-oriented and they are permanent.

Architectures containing this kind of an entity are typically the most similar ones to our approach. These include EMAA [13], which has servers providing services, as well as Hermes and Mole [2] with ServiceAgents. Also docks in EMAA have some similarities with our transporter, but distinctively our transporters only handle things related to the communication over network. This makes the architecture clearer and reusable, since if many communication protocols are needed, an area can contain several transporters of different types. Also our approach does not rely on the need for each node/transporter to be able connect to all other areas or to a centralized naming directory/resource server. On the contrary, the architecture model can be built in a way that the transporters work like routers and only know the next destination while asked for a certain type of a service. This is beneficial in several cases, for example, if communicating through several firewalls.

8. Concluding remarks

In this paper, we presented an agent based architecture style and specified it in three meta-levels. We also showed how higher level of abstraction in dependencies, and agent based communication are a feasible solutions in application integration. We validated our approach by implementing a framework for agents and by using it to create the architecture for a process support environment. We also showed in an example, how the way of specifying the architecture can be used also in specifying reusable architectural patterns (observer pattern example).

The presented architecture style attains a relatively good level of flexibility, customizability, and maintainability, as well as provides means for incremental development, e.g. because of the easiness of adding new entities to the system and keeping each business logic case in a single place. The architecture style is also simple, concrete, and well defined. There are some similarities to existing architectures, including other agent-based architectures and SOA.

The architecture model and the implementation of the case study could be improved and extended in many ways. For example, graphical specification of architecture meta-model combined with code generation facilities, as well as simple mechanisms for defining at least the simplest agents, like in BPEL (Business Process Execution Language) for Web services [3], might be useful. More extensive practical tests about performance, suitability, etc. would help us to understand all the benefits and disadvantages concerning the architecture model. We also continue the work with the implementation of the framework, for example, the case of adding new entities to the system could be more automated.

Acknowledgements

We would like to thank Kai Koskimies and the anonymous referees for their valuable comments. This research has been financially supported by the Institute of Software Systems (Tampere University of Technology), the National TechnologyAgency of Finland (project Inari), Nokia, Plenware Group, TietoEnator, and John Deere.

References

[1] J. Airaksinen, K. Koskimies, J. Koskinen, J. Peltonen, P. Selonen, M. Siikarla and T. Systä. xUMLi, towards a tool-independent UML processing platform. In Proceedings of 10th Nordic Workshop on Programming and Software Development Tools and Techniques (NWPER'2002), Copenhagen, Denmark, August 2002.

[2] Baumann J., Hohl F., Rothermel K., Straßer M., Mole – Concepts of a mobile agent system, World Wide Web 1 (1998) 123–137.

[3] Business Process Execution Language for Web Services Version 1.1, http://www-106.ibm.com/developerworks/webservices/library/ws-bpel/

[4] F. Buschmann et al. Pattern-Oriented Software Architecture: A System of Patterns. Wiley, 1996, p. 339-343.

[5] Chappell D., Enterprise Service Bus, O'Reilly, 2004.

[6] Corradini F., Mariani L., Merelli E., An agent-based approach to tool integration, International Journal on Software Tools for Technology Transfer (STTT), Volume 6, Issue 3, Aug 2004, Pages 231 – 244.

[7] Corradini F., Merelli E., Hermes: Agent-Based Middleware for Mobile Computing, Lecture Notes in Computer Science, Volume 3465, Jan 2005, Pages 234 – 270.

[8] Gray R., Cybenko G., Kotz D., Peterson R., Rus D., D'Agents: Applications and performance of a mobile-agent system, Softw. Pract. Exper. 2002; 32:543–573002E.

[9] Hohpe G., Woolf B., Enterprise Integration Patterns: Designing, Building, and Deploying Messaging Solutions, Addison-Wesley, 2003.

[10] Keen M., Acharya A., et al., Patterns: Implementing an SOA using an Enterprise Service Bus, IBM Redbooks, 2004, http://www.redbooks.ibm.com/redbooks/pdfs/sg246346.pdf

[11] Lange D., Oshima M., Mobile agents with Java: The Aglet API, World Wide Web 1 (1998) 111–121.

[12] Lange D., Oshima M., Seven Good Reasons for Mobile Agents, Communications of the ACM, March 1999/Vol. 42, No. 3.

[13] Lentini R., Rao G., Thies J., Kay J., EMAA: An Extendable Mobile Agent Architecture - AAAI Workshop on Software Tools for Developing Agents, 1998.

[14] Manvi S., Venkataram P., Applications of agent technology in communications: a review, Computer Communications 27 (2004) 1493–1508.

[15] Martin D., The Open Agent Architecture: A Framework for Building Distributed Software Systems, Applied Artificial Intelligence, 1999.

[16] Müller J., Bauer B. and Berger M., Software Agents for Electronic Business: Opportunities and Challenges, Lecture Notes in Computer Science, Volume 2322, Jan 2002, Page 61.

[17] Object Management Group, OMG-Meta Object Facility, v. 1.4, April 2002.

[18] Papazoglou M, Service-oriented computing: concepts, characteristics and directions, Web Information Systems Engineering, 2003.

[19] J. Peltonen, "Visual Scripting for UML-Based Tools", In Proceedings of ICSSEA 2000, Paris, France, December 2000.

[20] J. Peltonen, P. Selonen. An approach and a platform for building UML processing tools. In Workshop on Directions in Software Engineering Environments (WoDiSEE 2004), Edinburgh, Scotland, May 2004, pp. 51-57.

[21] Schoeman M., Cloete E., Architectural components for the efficient design of mobile agent systems, In proc. of SAICSIT, 2003.

[22] Shen, W., Norrie, D.H., Agent-Based Systems for Intelligent Manufacturing: A State-of-the-Art Survey. Knowledge and Information Systems, an International Journal, 1(2), 129-156, 1999.

[23] Sycara K., Paolucci M., Velsen M., Giampapa J., The RETSINA MAS Infrastructure, Autonomous Agents and Multi-Agent Systems, Volume 7, Issue 1 - 2, Jul 2003, Pages 29 – 48.

[24] Zimmerman O., Milinski S., Craes M., Oellermann F., Second generation web services-oriented architecture in production in the finance industry, OOPSLA, 2004.

[25] Zimmerman O. et al. Service-Oriented Architecture and Business Process Choreography in an Order Management Scenario: Rationale, Concepts, Lessons Learned. OOPSLA, 2005.