Advantages of Multi-agent Approach to Building of Monitoring Systems

Andrej Lúčny
Department of Applied Informatics, FMFI
Comenius University, Mlynská Dolina, 842 48 Bratislava, Slovakia
e-mail: andy@microstep-mis.com

Abstract

We introduce Agent-Space architecture which is a specific application of multi-agent paradigm. It is dedicated for building of systems which are not necessarily distributed but their agent-based organization brings a profit. In our experience, monitoring systems are very suitable to be implemented in this way. We demonstrate how we employed the architecture for their development and we discuss benefits which we have got. Namely we improved real-time operation, reliability, configurability and ability to be modified.
Keywords: agent, multi-agent system, agent-space architecture, real-time, monitoring.

1. Introduction

Within the last decade the technology-push research at the field of multi-agent systems has brought many more or less profitable applications at various domains. Though multi-agent modularity is traditionally dedicated for distributed systems, it is possible to apply it on any platform where we have more processes or threads. In our experience the real-time platforms for development of monitoring systems are very good candidates for multi-agent approach. We present a particular architecture which we have applied at this field and discuss how much it is profitable.

2. Multi-agent approach

Under multi-agent approach we understand any kind of system development which is based on multi-agent modularity. The best example is development of robotic team playing robot-soccer. It would be too difficult to write a program which controls all the players. It is much easier to code programs for individual players and let the team control to emerge from their interaction. Such interacting programs are called agents and can be implemented as objects equipped with an own thread of control and a mechanism of a mutual data exchange including sensation and action of the system environment. Each agent is endlessly running a sense-select-act cycle. Any course through this cycle calculates some actions upon information sensed from environment or provided by other agents. The communication mechanism can be based on direct message passing or on indirect communication through a more or less sophisticated blackboard.

A highlight feature of this kind of modularity is decentralization. Though one agent can depend on data which are provided by another agent, it does not mean that this agent stops its activity when we remove the other agent from the system. Any agent has own thread of control and does not need to receive a data to become active. Any such dependence means just that the outer behavior of the system can become more or less successful when we add or remove an agent from the system. For instance - concerning the robot-soccer example - when we remove midfielder from the team, striker does not stop (endlessly waiting for a pass from the removed midfielder), just probably scores a goal less frequently.

3. Traditional architectures of monitoring systems

As monitoring systems need to operate in real time, their traditional architectures are based on process architectures of real-time operating systems like QNX. Mostly it is based on blocking message passing and client-server relationship between communicating processes organized in a pyramid layout.

We can demonstrate the traditional architecture on the following example: There are two places where a floating average of a quantity measured at the first place should be displayed. Therefore we put a probe on the first place and interconnect the two places by two communication lines (to have backup). Then we can decompose the system into the following processes (Figure 1):

- *driver*, which performs measurement from the probe device and which is able – as a server – to provide the last measured value to any client which sends it a proper request.
- *average*, which is a client of *driver*, regularly asks it for measured values and acts in parallel as a server which provides their floating average
- *display*, which is a client of *average* and just displays the floating average on a specific display device. Thus we have displayed the required value at the place of measurement. Of course, we re-use the same process for the same job on the second place.
- transmission of the required value from the first side to the other side can be implemented by a pair of processes which operate over the communication lines. *sender* is a client of *average* and put each value got from *average* to the line which it controls. *receiver* receives the value from the line and provides it at request to any process on the other side.
- Having one line, we would be able to implement *receiver* to have the same client interface as *average*. Thus we could let *display* to get values directly from *receiver*. Unfortunately, we have two lines and thus we need to implement *average2* which is a client of both *receiver*-s and provides transferred values to *display* regardless they are received from the first, second or both *receiver*-s.

This kind of architecture is widely used and it is fine for any usual need, mainly if the requirement is such simple as in our example. However having necessity to build complex systems operating under various and uncertain conditions, open to customization and later modifications, the architecture exhibits several disadvantages.

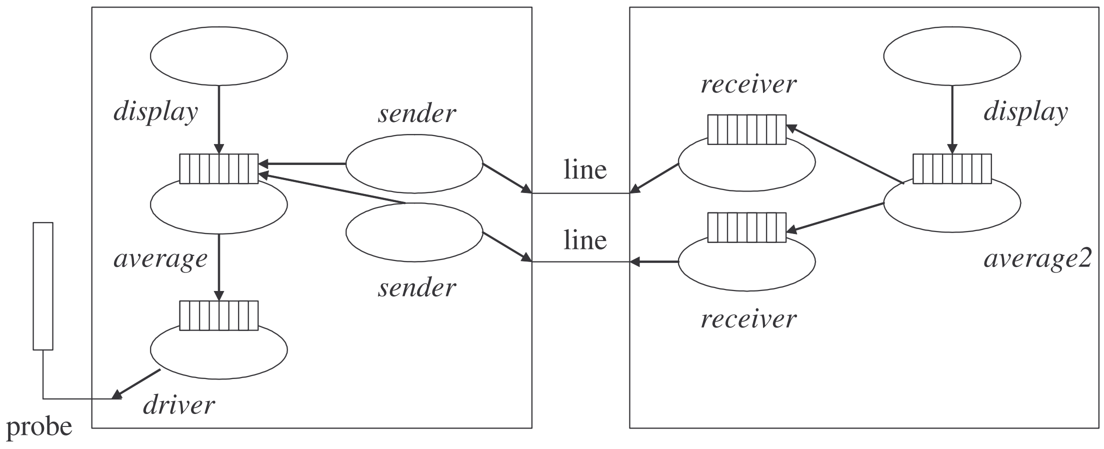

Figure 1. Example of traditional architecture

In this paper we focus on two of them:

1. it is very difficult to restart just a part of such system, because restart of any server process requires restart of its client processes and these processes acts as servers for other clients, …, etc. Finally, we need to restart almost whole the pyramid.
2. it is impossible to modify existing data flows without modification of existing processes.

Why we need to treat these problems? Well, at the first, it is not possible to implement a complex system without hidden errors and without necessity to adjust its parameters or install updates during its course. For all of this we need the restarts. At the second, complex systems are developed incrementally. And the best method of incremental development is to add increments without any modification of the former parts. Or course; in this case, we need at least the ability to modify data flows among the former parts.

4. Agent-Space architecture

It is obvious that the both above mentioned problems have something to do with decentralization. Therefore multi-agent modularity could help us to overcome them. One possible solution is application of Agent-Space architecture proposed by us in [2]. The architecture is based exclusively on indirect communication, i.e. agents just read and write messages on a common blackboard (called *space*). The messages are referenced by name and a newer value potentially overwrites any former value of the same name. An important feature of these messages is their bounded time validity, i.e. a writing agent can define a period after which the written content disappears. Besides it, any value can have associated priority which protects it against overwriting by values with lower priority. However agent which has written the lower-priority value is not aware of the fact that the value has never been written.

Further it is important to explain that activity of the agents is not data-driven. Agents are invoked to perform their sense-select-act code mainly by timer and even in case of invocation by trigger (a change in the *space*), the invoked agent knows just that something has changed, but no particular data are routed to it.

Concerning the above example, we can make an analogous solution under Agent-space architecture (Figure 2). We will use two names of messages in *space* (on the blackboard):

- ***current*** for storage of the current measured value
- ***average*** for storage of the floating average

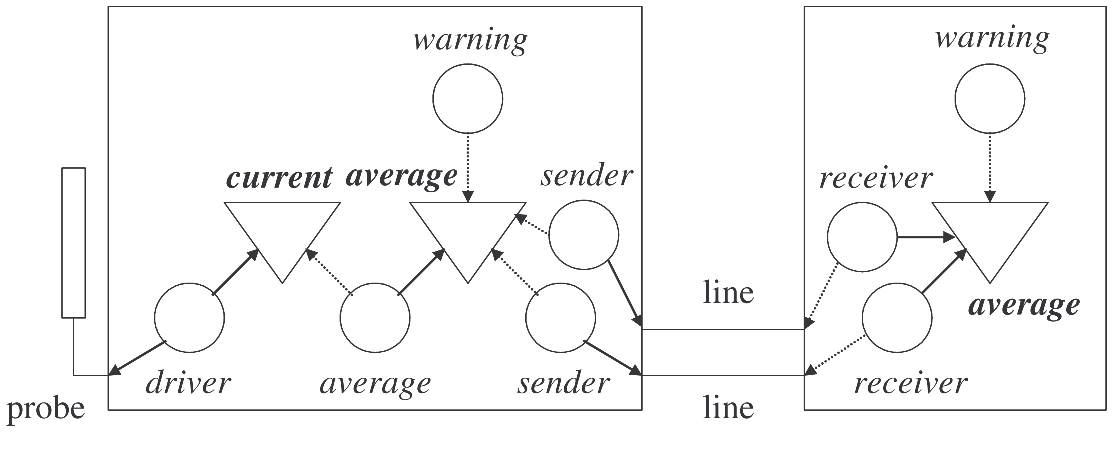

Figure 2. Example of Agent-Space architecture

Further we will launch three agents at the place of the measurement:

- *driver* which measures from probe and writes the measured values to ***current***
- *average* which reads values from ***current*** and writes calculated floating average into ***average***
- *display* which reads values from ***average*** and displays

In this moment we have displayed the required value at the place of measurement. Transmission of the value to the other side can be implemented by pair of agents – *sender* and *receiver*. While *sender* is polling values from ***average*** and transmits them to the line, *receiver* gets them from the line and writes into ***average*** message on the other side. It is interesting that both *receiver*-s can write into the same message as they write always the same. Just *receiver* must not write something like *'bad value'* when its line is broken; rather it has to be quite in this case. Of course, then there is a danger that both lines become broken and the last value of ***average*** remains displayed forever. Therefore all such agents as *receiver* have to write their values with bounded time validity. Thus we do not need to implement an agent analogous to *average2* from the traditional solution. Finally, *display* agent is re-used on the other side to display the transmitted value.

Now imagine what happen when one communication line is fast and the other one very slow. The *sender* operating over the slow line spends too much of time by writing data to the line. Thus it is not able to undertake every value of ***average***, just every third one, for instance. The system does not handle this situation as an exception; it is concerned to be correct. In this way Agent-space architecture supports real-time operation.

5. Reliability, configurability and soft crash landing sybsystem

Now, let us compare the two above mentioned architectures according to the problem one, i.e. the problem with restarts. Concerning our example, imagine that the probe generates occasionally such response that *driver* crashes. Usual way how to treat such states is to establish so-called soft crash landing subsystem which follows course of our system and tries to recover it when something crashes. Such subsystem can easily detect that *driver* has crashed and launch it again. However, concerning the traditional solution, *average* looses connection to the *driver* by this act, thus it also needs to be restarted. Analogically, the same is required for *display*, and both *sender*-s. Therefore we can rather restart whole the system. Thus the soft crash landing subsystem is reduced to a pure watchdog. However, within agent-space architecture, while the *space* (blackboard) is operational, it is always enough to restart one agent.

Of course, after such restart, still it can happen that a transaction opened by the former instance of the restarted agent is corrupted. Such corruptions have an interesting relation to well-known problem in multi-agent paradigm, called accessibility of internal state. If it is not allowed to agents to keep information in their internal state – in other words they are forced to keep it in *space*, any kind of transaction can continue after restart without any corruption.

6. Ability to be modified - incremental development

Regarding the second problem – i.e. possibility to modify data flows without modification of the former codes – imagine that we would like sometimes to display a manually entered value instead of the measured one. (It is a typical requirement for any automatic monitoring which is verified by observation.) Within the traditional solution we have to modify *average* to be able to undertake the manually entered value instead of polling *driver*. There is no other option. However, within agent-space solution, we can simply add an agent which put the manual value directly into ***average*** in *space*. Just it has to write it with a higher priority to protect its overwriting by new values from *driver*. Consequently, the agent must remove the value when it is required to renew displaying of the values provided by measurement. Alternatively, the agent can specify bounded time validity for the manual value when it writes the value into *space*.

7. Conclusion

In this article we have compared the traditional architecture of monitoring systems with a novel solution based on multi-agent approach. We discussed advantages of such solution; namely we focused on reliability provided by soft crash landing subsystem, configurability and ability to be modified.

References

[1] Kelemen, J.: *The Agent Paradigm.* Computing and Informatics, Vol.22. (2003), pp. 513-519

[2] Lúčny, A.: *Building Complex Systems with Agent-Space Architecture.* Computing and Informatics, Vol. 23 (2004), pp. 1001-1036

[3] Lúčny, A.: *From inter-module links to indirect communication among agents.* ZNALOSTI ‘07, VŠB Ostrava, 2007