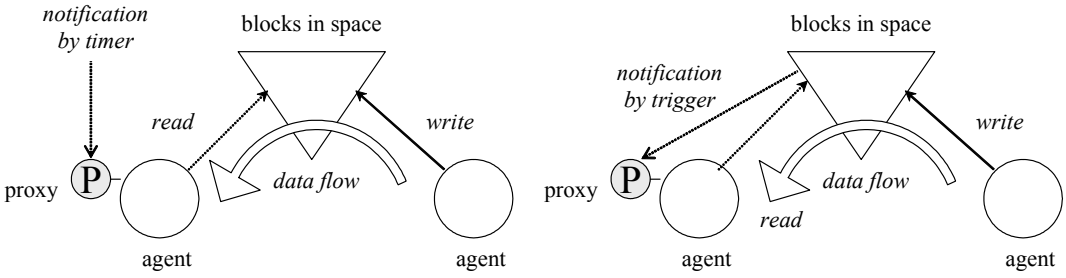
Refinement and emergency versus design and predetermination

Andrej Lúčny

Institute of Informatics FMFI UK Bratislava and MicroStep-MIS
e-mail: andy@microstep-mis.com
web: http://www.microstep-mis.sk/~andy

Keywords: reactive agent, space, refinement, emergency, multi-agent systems

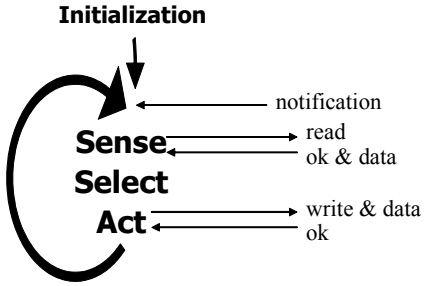
Here we summarize our experience with building systems based on reactive agents and their interaction via indirect communication. Our primary aim was to develop monitoring and control systems. Because of their real-time nature we selected a QNX4 platform. This platform provided us not only with sound real-time (with latency 2μs) which run on usual machines like PC, but also with multitasking which enables hundreds of processes and a unique data exchange model based on blocking message passing[1]. At this point, we encountered difficulties (like full-duplex data exchange between two servers) with factory-recommended architecture (so called pyramidal client-server architecture), and for that reason since 1996 we have been using our own architecture based on non-blocking message passing and indirect communication. Unlike the usual solutions of non-blocking message passing, which are based on the communication through a special server called a *message queue*, we used an alternative approach based on another special server called a *space*. This server has an ability to contain named data drops (called blocks) with given time validity and its clients can write to blocks and read them as well as they can get a notification about their change (called a *trigger*)[2]. (The last mentioned service is not inevitable since clients are usually woken up by the notification from a *timer*. However, it can be very useful when the data, which has to be processed swiftly, is coming in irregular instants of time and there is not enough power to cover the redundancy of analogical regular processing.) As a result, each block can be shared by more producers and consumers (this is the difference in comparison to the concept of the message queue).



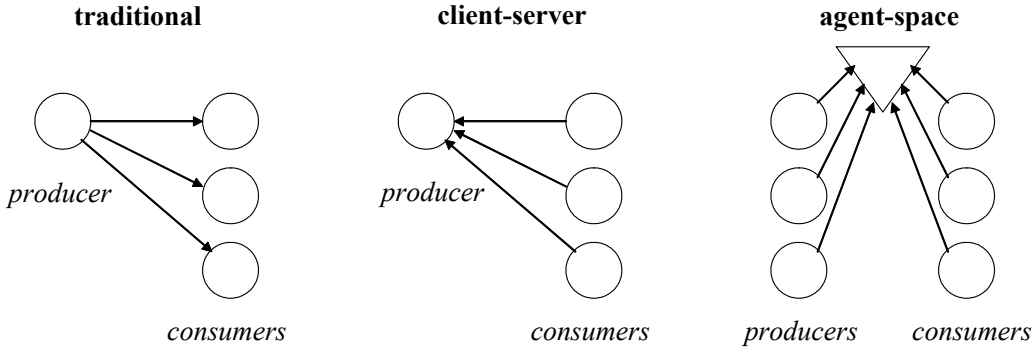
In this way we eliminated all the difficulties associated with the pyramidal

[1] So called SRR model
[2] This concept of communication is sometimes called stigmergic communication and as we found later, it is similar to a type of data exchange, known as LINDA tupple space, which can be found in a parallel programming domain.

client-server architecture, as the *space* became the only server in the system. All other process turned to clients of *space* and their natural form lay in a simple sense-select-act cycle (reading from *space*, computing reaction, writing to *space*) activated by a notification from a *timer* or a *trigger* (and in special cases also from a system device or from the user interface). We realized that this concept is in fact a kind of application of a certain multi-agent system (i.e. distributed system) for building single-node multi-process systems, conformable to slogan "computer is a network". Therefore we designated the *space* clients as *agents*. Further, because of the simplicity of the select phase, we attributed to them the characteristic adjective: *reactive*.



This architecture provided us with many advantages (though it also brought some problems as the absence of 100% reliable mechanism of synchronization among processes). The most important advantage was free communication among producers and consumers.

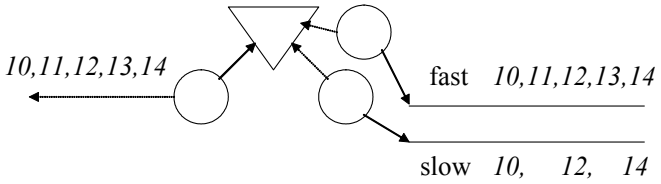


Consequently it was possible to restart a producer without any impact on its consumer (useful for recovery from errors), or replace a producer by another one (useful for ability to configure). The next positive (though controversial) feature was implicit sampling of produced data when a consumer was too slow to undertake them all. This sampling is based on the fact that producers overwrite data in blocks regardless consumers have read them or not[3]. Thus the real-time operation is supported: the data, which cannot be processed quickly enough, is lost[4].

---

[3] On the other hand, if a consumer receives more notifications during single course through its cycle, they are handled as a single notification.

[4] This feature is unacceptable for some applications (like counting money) and very profitable for other domains (like control systems of mobile robots). In practice, its negative impact is usually eliminated by real-time (so the feature and real-time support each other).



Additional benefit resides in the separation of a domain-dependent code from a data-exchange code and in normalization of data-exchange interfaces (reuse-ability).

However, the most interesting change happened with the development process. As our systems have usually tens sometimes even hundreds of versions, the ability to modify is a crucial factor for us. In fact, this ability improved when we expressed the global system behavior by implementation of a local behavior of many reactive agents[5] manipulating with blocks (and in special cases also with system devices and the user interface). Of course, a typical modification was usually realized by one of the following ways:

- by adjustment of certain agent codes
- by development of new agents which alternate some former ones
- by development of some additional agents which profit from the former agents but have no impact on their operation (they implement additional features only)

However, we encountered also cases when a much more advantageous choice was:

- to develop new agents which are able to influence behavior of the former ones in a proper manner (without an adjustment of their code).

This influence is realized by writing to former blocks to which some former agents are "sensitive"[6]. This strategy is effective even in the case of the correction of mistakes.

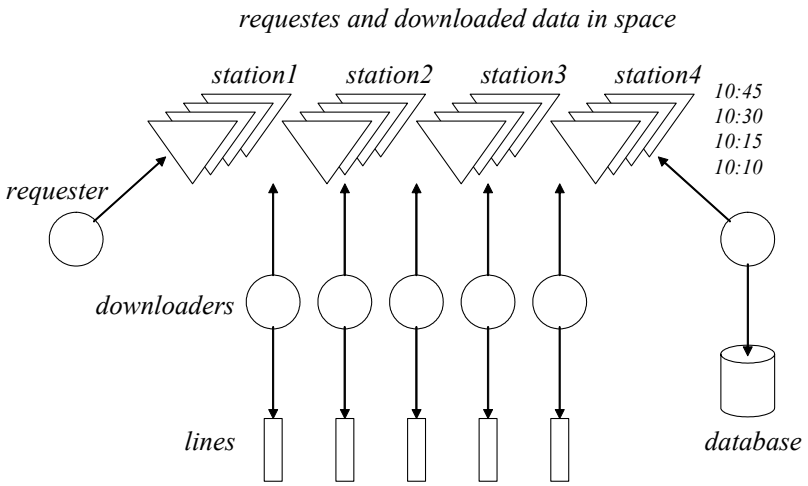
Let us demonstrate it on a live example. We had to develop a system which collects periodic data (available without gaps in the centre as soon as possible). The system was supposed to call through several dial-up PSTN lines to get data from many outstations connected to the GSM network. Under such physical conditions the collection is quite unreliable for many reasons (the overloaded GSM network, overloaded router between PSTN and GSM or noise in the GSM network). On the other hand, the outstations (running on a solar panel and an accumulator) are almost reliable and contain data for 99% of the measuring periods. Thus it was possible to download most of the data to the center; however the system had to take into consideration that an individual download can fail owing to connection failures and interruptions in any moment of data retrieval or due to noise on a line (corrupted request or corrupted response). Of course, the system had to be resistant against such exceptional states as the failure of some PSTN lines or the absence of data on a new station. Moreover, the amount of PSTN lines was much less than the amount of outstations and the duration of lines occupation had to be as short as possible.

We designed the system as a complex of a *space* server and reactive agents. The *space* contained data requests for a particular station and time period, locks for individual stations and downloaded data. There was an agent called a *requester* which was invoked at the end of each period and it generated requests for that period (each

[5] Regarding programs, we used tens of agents. Regarding processes, their number is from tens to hundreds.

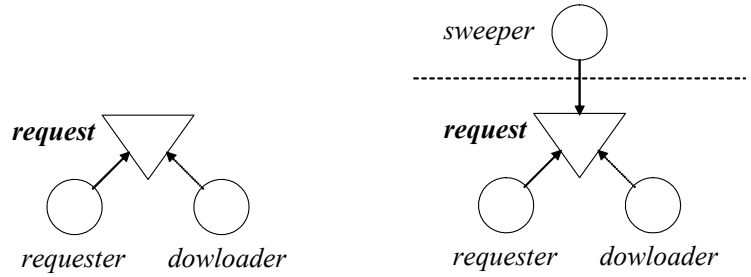
[6] This trick is quite similar to Brook's subsumption architecture, however unlike his *suppressor* and *inhibitor* our kind of influence is not so deterministic: despite the writing to a block by a new agent, the former producers are still active and compete with it.

request had certain long-term validity e.g. one day). Further each PSTN line was controlled by an agent called a *downloader*. These *downloaders* were regularly invoked and each of them was supposed to find in the *space* an unanswered valid request. If such a request was found, they tried to lock the station associated with it[7]. In the case the station had not been locked yet, the agent collected all requests related to the station, dialed the station and after successful connection it tried gradually to download data for each requested period. Those requests, for which the data was downloaded correctly (regarding CRC) or a correct *not available* response was received, were removed from the *space* whereas the validity of others was adjusted in the following way: its start was shifted to the next time period and the end remained as it was. Thus after a certain period chronically unsuccessful requests became invalid and they were removed from the *space* automatically. Finally, there was an agent which undertook downloaded data from the *space* and stored them in a database.

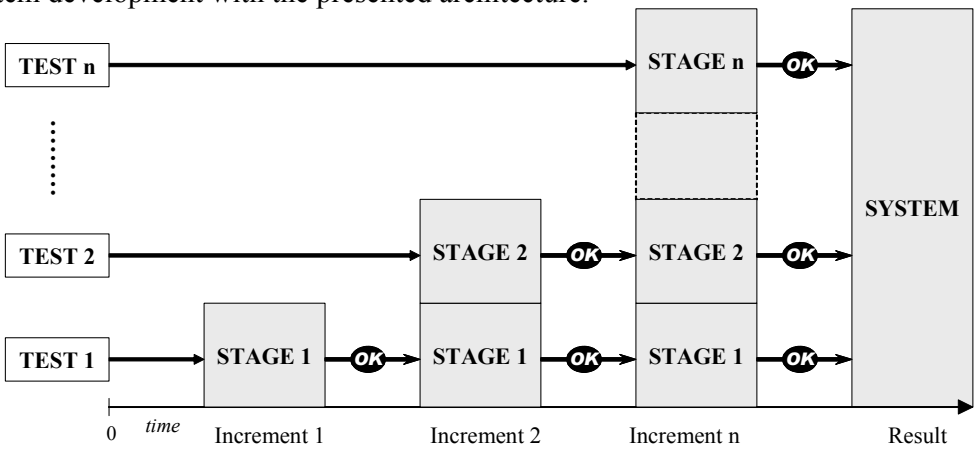


The system was running in a good way, but we found that it downloaded only 97% of the data in spite of the fact that long-term log-files at outstations contained more than 99%. It was clear that there had to be a mistake in the implementation of the *downloader* since its design should theoretically guarantee downloading of all the data available at the stations. This situation was probably caused by an unknown factor which was not taken into consideration. Logically we had to reveal this factor and adjust the code of the *downloader*. However, we did not. It was much easier to add another agent called a *sweeper*. At the end of a day, this new agent generated requests for missing data for those stations which data was almost complete but still contained some gaps (regardless the fact that some of this missing data could have been already testified as *not available*). In this way the *sweeper* intruded into the job of the *requester* and forced *downloaders* to regard these requests as generated by the *requester*. As a result of this modification, data completeness achieved the expected 99%. Thus we developed a correct system, though it consisted of two not completely correct parts.

[7] To realize a lock, we need ability to perform more uninterrupted operations over space; particularly in this case one read and one write operation.



In general, instead of regarding the design of local behavior of individual agents, it is sometimes more effective to examine the global behavior of the whole system (i.e. how blocks are changing in the *space*) in the way as if we lost control over the system. Consequently the derived modification is nothing but an ordinary patch which could even insult our belief of how computer systems should be constructed. In fact, such modifications represent more refinement than design and its correctness cannot be logically derived, only tested after implementation. However, why should not we accept a more effective strategy? I think the discussed strategy is the most natural one for system development with the presented architecture.



Finally, why should we use such an architecture? Just take into consideration in our example how implicitly and at no cost we achieved a resistance of the system to such states as a PSTN line failure or excess of outstations in the network (when the system operates, only the data presence in the centre is delayed). These features simply "emerged" thanks to the nature of the mentioned architecture.

[1] Brooks R.: *Cambrian Intelligence.* The MIT Press, Cambridge, Massachusetts, 1999
[2] Valckenaers P., Van Brussel H., Kollingbaum M., Bochman O.: *Multi-agent Coordination and Control Using Stigmergy Applied.* In: Multi-Agent Systems and Applications (Luck M., Mařík V., Štěpánková O. Trappl R., eds.), ACAI, Praha, 2001
[3] Kelemen J.: *From statistics to emergence -- exercises in systems modularity.* In: Multi-Agent Systems and Applications (M. Luck et al., eds.). Springer, Berlin, 2001
[4] Lúčny A.: *Spaces and reactive agents under QNX4.* QNX Tools and Technologies, Bratislava, 2001
[5] Waldo J.: *Mobile Code, Coordination and Changing Networks.* Concoord, Lipari, 2001