Chapter 3

Agent Communication

Agent-to-Agent and Agent-to-World Communication Models

This chapter introduces some communication and interaction aspects of MAS and the relationship to the *AAPL* agent behaviour model. The *AAPL* tuple-space agent interaction is presented and its relation to mobile processes and the Π-Calculus with channel-based communication.

3.1 Shared Memory

The shared memory model is a commonly used inter-process communication paradigm for parallel, rather less for distributed systems. It is closely related to the parallel register and random access machine (PRAM), shown in the Figure 3.1 below, and discussed in Chapter 5 in conjunction with the Communicating Sequential Processes (CS) data processing model.

The PRAM model assumes *n* identical processing units (PU), which are connected to a shared memory resource with a random access model (SRAM), with memory cells of equal width that are referenced by a numerical address. The SRAM model supports read and write operations. Based on the RAM access model (Exclusive or Concurrent Read and Write), one or more PUs can access the SRAM. Concurrent read operations are in principle free of conflicts (neglecting technological and architectural constraints and conflicts), but write operation can cause access conflicts, e.g., writing of multiple PUs to the same memory cell storing different values. These conflicts require a conflict resolution function, commonly solved by mapping concurrent access on a sequential ordered exclusive access model. The SRAM and PRAM model provides no explicit synchronization between different PUs and processes they are executing, and it introduces a strong coupling of the PUs. A shared memory model is attractive in parallel and distributed programming because it offers a simplified parallelization paradigm for existing algorithms, which can be supported by programming language extensions, e.g., multi-threading extensions for established programming languages, or in distributed programming languages like *Orca* [BAL90]. In distributed systems the reading, writing, and updating of cached SRAM cells are encapsulated in message passing. Without additional synchronization primitives the SRAM model is not useful for inter-process and inter-agent communication. Furthermore, the SRAM model "violates" the autonomy feature of agents and the loosely coupling of agents with their environment.

This model is used by the *SeSAm* MAS simulator exclusively for the implementation of the inter-agent communication. The simulator is used in this work for all kinds of behavioural MAS and agent platform simulations, discussed in Section 11.

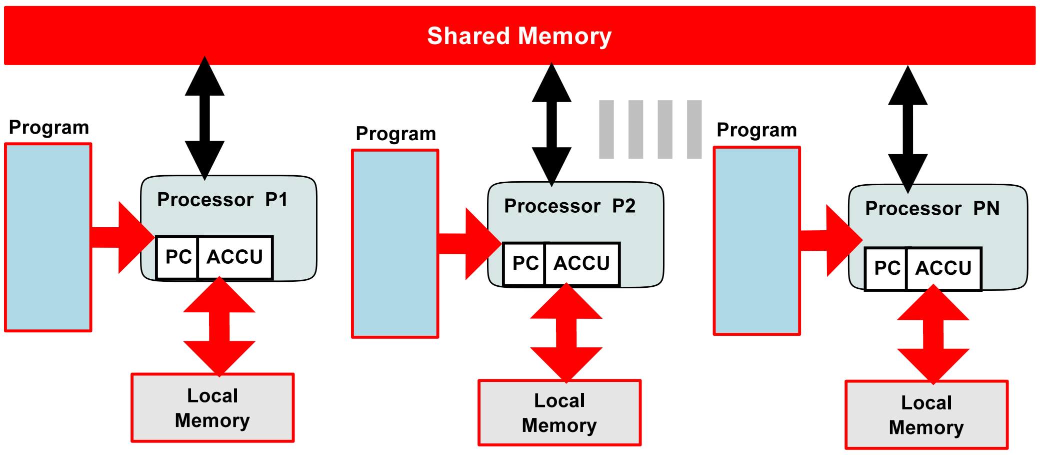

Fig. 3.1 *The Parallel Random Access Machine (PRAM)*

3.2 Tuple Space Communication

Tuple spaces represent an associated shared memory model, considering the shared data as objects with a set of operations supporting the access of data objects, which are organized in spaces that can be considered as abstract computation environments. A tuple-space connects different programs, which can be distributed if the tuple-space is distributed, too, or at least its operational access. One well-known tuple-space organization and coordination paradigm is Linda [GEL85]. The tuple-space organization and access model offers generative communication, i.e., data objects can be stored in a space by processes with a lifetime beyond the end of the generating process.

3.2.1 The Data Model

The data is organized with tuples. A tuple is a loosely coupled compound of an arbitrary number of values of any kind. A tuple is a value and once stored in a tuple space it is persistent. Tuple types are similar to record types, but they are dynamic and can be constructed at run-time on the fly without any static type constraints. The field elements of tuples cannot be accessed directly, commonly requiring pattern matching and pattern-based decomposition, in contrast to record types offering named access to field elements, though treating of tuples as arrays or lists can solve this limitation. A tuple with *n* fields is called n-ary.

Formally, tuples are defined as vectors by the following generation rule with values *v*, expressions ε, and variables *x* that are considered as actual parameters (i.e., variables *x* used with value semantics):

$$e = \left\langle \vec{d} \right\rangle, \text{ with } \vec{d} ::= d \mid d;\vec{d} \text{ and } d ::= v \mid \varepsilon \mid x$$

Tuple values require pattern matching based on template pattern, with the following generation rule, consisting of actual (v,ε,*x*) and formal parameters (*x*?, variables used with reference semantics):

$$t = \left\langle \vec{dt} \right\rangle, \text{ with } \vec{dt} ::= dt \mid dt;\vec{dt} \text{ and } dt ::= v \mid \varepsilon \mid x \mid x? \mid \perp$$

A search pattern can use a wild-card (⊥) instead of formal parameters. Each tuple *e* has a type signature *Sig*(*e*) = $S_e = \langle T_1; T_2; \ldots ; T_n \rangle$, a tuple of the same arity as *e*, specifying the type of each tuple field. A tuple can only be addressed by its association with templates.

Def. 3.1 *Matching rule for tuples and templates*

Let e = $\langle d_1; d_2; \ldots ; d_n \rangle$ be a tuple, t = $\langle dt_1; dt_2; \ldots ; dt_m \rangle$ be a template; than e matches t (denoted by $e \succ t$ and match(e,t)=true) if the following conditions hold: (i) m = n. (ii) $\forall dt_i = d_i$ or $dt_i = \perp, 1 \le i \le n$. Condition (1) checks if e and t have the same arity, whilst (2) tests if each non-wild-card field of t is equal to the corresponding field of e.

Commonly the first field of a tuple is handled as a symbolic key identifying a tuple subclass by using text strings or enumerated symbolic constant values.

3.2.2 The Operational Semantics

There is a set of operations that can be applied by processes, consisting of a set of pure data-access operations treating tuples as passive data objects, and operations treating tuples as some kind of active computational objects (more precisely, data to be computed), offering Remote-Procedure-Call (RPC) semantics.

out(e)

The execution of the output operation inserts the tuple *e* in the tuple space. Multiple copies of the same tuple value can be inserted by applying the output operation iteratively. The equal tuples cannot be distinguished after insertion in the tuple space.

Examples:

```
out("Sensor",1,100);
out("Sensor",2,121);
```

in(t)

The execution of the input operation removes one tuple *e* from the tuple space matching the template *t*.
Examples:
```
in("Sensor",1,s1?);
in("Sensor",i?,s?);
```

rd(t)

The execution of the read operation returns a copy of one tuple *e* matching the template *t*, but does not remove it.
Examples:
```
rd("Sensor",1,s1?);
rd("Sensor",i?,s?);
```

inp(t),rdp(t)

Non-blocking versions of the read and input operations, discussed in the synchronization model of tuple spaces below. They can be used to test and read/input a tuple.
Example:
```
res=inp("Sensor",1,s1?);
```

in?(tmo,t),rdp(tmo,t)

Not permanently blocking versions of the read and input operations, discussed in the synchronization model of tuple spaces below. They can be used to test and read/input a tuple.
Example:
```
res=in?(0,"Sensor",1,s1?);
```

eval(τ)

This operation allows the injection of tuples that are currently not fully evaluated using an extended functional tuple τ (with extended $dt::=v|\varepsilon|x|f(x)$ with a function argument). This operation assumes a function $f(x)$ that is present in the processes participating in the tuple space and accessible to the tuple space client implementation, which cannot be related to the *AAPL* agent behaviour model.
Therefore, a different approach is applied. A partially (active) tuple, which was stored in the tuple space by a client process, is consumed by a service process that executes the function(s) $f(x)$, finally replacing the partial tuple with a fully evaluated tuple. For this purpose, the process providing the service uses the *listen* function to receive evaluation requests and the *reply* function to pass the evaluated tuple back to the requesting process.

Example:

```
P1: eval("square",2,y?)
P2: def sq = fun x -> x*x;
    trans := listen("square",x?,?);
    y := sq(x);
    reply(trans,"square",x,y);
```

The *listen* function provides a transaction reference that can be used by the *reply* function to address the destination of the tuple.

3.2.3 The Synchronization Model

There are producer (generator) and consumer processes. A producer generates a tuple that can be withdrawn by a consumer process. The tuple output operation terminates immediately (asynchronous), alternatively after the tuple was stored in the tuple space (synchronous). A consumer process is blocked if the request cannot be serviced because there is actually no matching tuple in the tuple space. After a matching tuple was stored in the tuple space, it will be immediately assigned to one of the waiting consumer processes. Therefore, the input operation is always synchronous. The only exception are the not permanently locking versions, limiting the waiting to an upper time bound (time-out).

There is no initial temporal ordering of producer and consumer operations. This should be illustrated in the following example.

Ex. 3.1 *Two producer and consumer processes performing multiple tuple space operations, and different possible tuple ordering outcomes*

```
P1 ⇐ out(ADC1,50); out(ADC1,100);
P2 ⇐ in(ADC1,x?); in(ADC1,y?);

P1 || P2 → P1 || (P2{50/x,100/y} |
                  P2{100/x,50/y})

P1 ⇐ out(ADC,1,50); out(ADC,2,100); out(ADC,3,200)
P2 ⇐ in(ADC,⊥,x?); in(ADC,⊥,y?);

P1 || P2 →            P1 || (P2{50/x,100/y} | P2{50/x,200/y} |
                              P2{200/x,100/y} | P2{200/x,50/y} |
                              P2{100/x,50/y}  | P2{100/x,200/y})
```

3.2.4 Distributed Tuple Spaces

Distribution of tuple spaces on different nodes implies synchronization issues, requiring usually reliable group communication that cannot be expected in technical sensing systems. Distribution of tuple spaces means the distribution and asynchronous execution of a set of tuple space servers, rather than one server. A tuple space server provides the necessary coordination for concurrent or interleaved in/out requests. Distribution of the servers leads to a distribution of coordination. But this issue can be solved by partitioning the tuple space in sub-spaces, as already introduced, and servicing each sub-space on a different node by one server. Coordination takes only place with tuples and templates of the same arity. A distributed search (of the n-th dimension server) and retrieval is remaining. But this approach lacks of sufficient "load" balancing because the tuples are usually not equally distributed with respect to their dimension. There are some work addressing the distributed search and retrieval (e.g., [ATK08]).

3.2.5 Distribution by Mobile Agents

Keeping the mobile *AAPL* agent model in mind it is not required to implement distributed tuple-spaces for inter-agent communication. Distribution of data is performed by the mobile agents themselves carrying data from one to another processing node, each providing a tuple space.

3.2.6 Markings and Garbage Collection

Tuples are persistent and exist in a tuple space until a tuple is removed explicitly. In the context of MAS this can cause to fill up of spaces with orphan tuples. To avoid this space leak, tuples can be marked with a lifetime tag limiting the existence of a tuple to specific time interval. A garbage collector can be used to iteratively age these markings until they reach zero lifetime, finally deleting these tuples. For this reason markings are called temporary tuples. A marking is defined by the following generation rule.

$$m = \left\langle \tau, \vec{d} \right\rangle, \text{ with } \vec{d} ::= d \mid d;\vec{d} \text{ and } d ::= v \mid \varepsilon \mid x, \quad \tau : \text{timeout}$$

3.3 Communication Signals

Signals are well-known asynchronous inter-process communication objects. *AAPL* signals can be propagated between agents beyond the local node scope, and can be considered as access paths to agents. Access paths in the Π-calculus are mobile channel names, corresponding to the mobile *AAPL* signals and agent identifiers, discussed in the next section.

3.4 Comparison: Signals and Tuples

Communication is central in large-scale distributed systems with a high impact on the overall system performance and stability. Two main issues arise affecting design considerations: 1. Efficiency of the communication with respect to latency, computational complexity, and storage; 2. Addressing and delivery of messages to destination entities. Efficiency is a key factor in self-powered material-integrated low-resource computing networks. Commonly, end-to-end communication is established between processes using IP protocols and computer nodes having unique addresses, which is not suitable for large-scale material-integrated networks and mobile agents. The agent model usually announces autonomy and loosely coupling to the environment, thus without a strong binding to a specific node. Moreover, there is usually no global knowledge of the current position of an agent in the network at all. Basically three approaches are available to exchange information between agents: Tuple-space access, signal messages, and agent migration.

Tuple-Spaces

Tuple-space communication (see Figure 3.2 a) exchanges data tuples between entities based on pattern matching, i.e., between processes and agents. The information exchange is data-driven and bases on the data structure and content of the data, and do not require any destination addressing or negotiation between communicating entities. Furthermore, tuple-space communication is generative, i.e., the lifetime of data can exceeds the lifetime of the producer. There are producer and consumer agents. A tuple-space can be localized with a limited access region, commonly limiting data exchange to entities executed on the same network node. Distributed tuple-spaces require inter-node synchronization and base on replication and some kind of distributed memory model. The *AAPL* agent model originally uses tuple-spaces only for agent communication executed on the same node.

Agent-to-Agent (A2A) Signals

Signals are lightweight messages that are delivered to specific agents (Agent-to-Agent A2A, see Figure 3.2 b), in contrast to the anonymous tuple exchange. One major issue in distributed MAS is remote agent communication between agents executed on different network nodes. Although an agent can be addressed by a unique identifier, the path between a source and destination agent is initially unknown. For the sake of simplicity and efficiency, routing table management and network exploration in advance is avoided. Instead, the *AAPL* platforms (*JAM*/*PAVM*) support signal delivery along paths of mobile agents only. That means, a signal from a source node *A* can only be delivered to a destination agent currently on node *B* iff the destination agent was executed (or created) on node *A* some time ago. I.e., two agents must have been executed on the same node in the past. Agent migration and signal propagation is recorded by the agent platform using look-up table caches with time limited entries and garbage collection.

Agent-to-Node (A2N) Signals

Previous agent platform implementations only support signal delivery along migration paths based on the destination agent identifier (private, uni-cast) or the agent class (public, broadcast). The new *JAM* platform 2.0 (see Chapter *8*) introduces signal delivery of signals to specific remote platforms (remote signalling) based on paths specified by the signal sender agent, shown in Figure *3.2* (c). The destination platform node broadcasts the signal to all listening agents executed on this particular node. To simulate private A2A uni-cast (or multi-cast) communication, agents can use a randomly generated signal name only known by the sender and the receiver. This new approach enables interaction between agents never executed on the same node. Furthermore, these remote signals are used to implement distributed tuple-spaces, discussed later.

Mobile Agents

Mobile agents can be used to distribute information in networks. They can get data/information from the tuple-space of the current node and store them in remote tuple-spaces by migrating to the respective nodes. The advantage of this approach is the ability to find suitable remote nodes and paths to nodes autonomously or based on content negotiation, and to filter, map, or process the collected data, e.g., using data fusion techniques. The disadvantage is a high communication and processing overhead due to the agent process migration.

Additionally, mobile agents can be used to deliver data in mobile network environments by using a mobile device for spatial migration (piggyback approach), not possible with A2A/A2N signals or remote tuples.

Distributed Tuple-Spaces

The new JAM platform 2.0 introduces the support for tuple migration using the `collect`, `copyto`, and store operations performed by agents. This feature enables the composition of distributed tuple-spaces controlled by agents. The collect and `copyto` operations transfer tuples from the local tuple-space to a remote using pattern matching, similar to the `inp` and `rd` operations. The `store` operation sends a tuple to a remote tuple-space, similar to the out operation.

Remote tuple space access is performed via A2N signals, shown in Figure *3.2* (d).

Evaluation

Figure *3.2* shows the analysis of simulation results as part of the use-case study [[BOS17B] for different agent communication strategies. In this simulation, sensor data derived from an artificial physical system, was distributed event-based, leading to varying activity patterns depending on the dynamic change of the mechanical structure under test within a given time window (15000 MAS simulation steps with 100 physical simulation iterations).

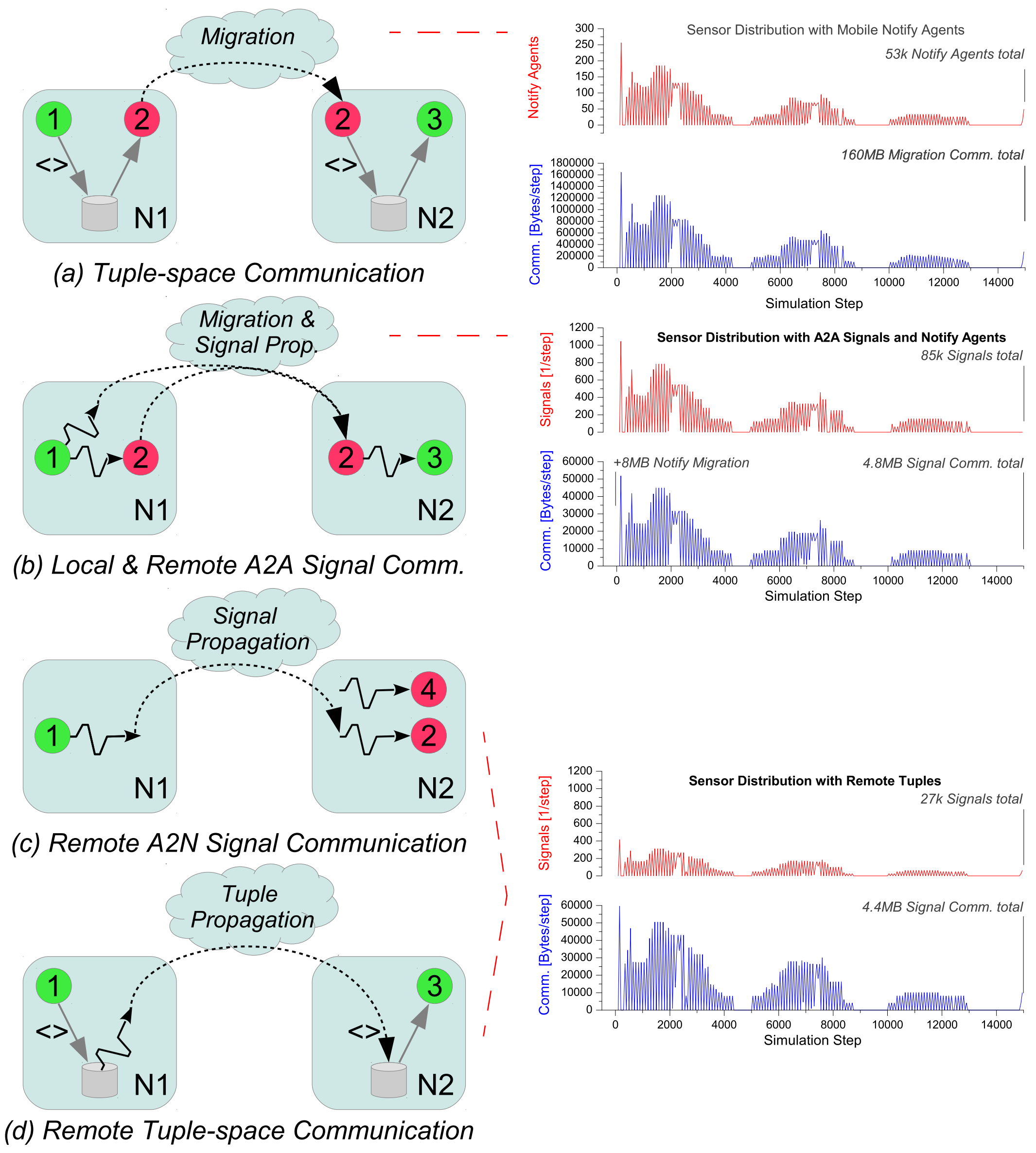
Migration
N1
N2
(a) Tuple-space Communication
Migration & Signal Prop.
(b) Local & Remote A2A Signal Comm.
Signal Propagation
(c) Remote A2N Signal Communication
Tuple Propagation
(d) Remote Tuple-space Communication
Sensor Distribution with Mobile Notify Agents
53k Notify Agents total
Notify Agents
160MB Migration Comm. total
Comm. [Bytes/step]
Simulation Step
Sensor Distribution with A2A Signals and Notify Agents
85k Signals total
Signals [1/step]
+8MB Notify Migration
4.8MB Signal Comm. total
Sensor Distribution with Remote Tuples
27k Signals total
4.4MB Signal Comm. total

Fig. 3.2 *(Left) Agent communication in AAPL (a) Tuple-space communication between agents on same node (b) Agent-to-Agent Signals (c) Remote Agent-to-Node Signals (d) Remote Tuple Operation (Right) Comparison of different approaches*

The Multi-domain simulator *SEJAM2P* was used to perform the evaluation and simulation providing real communication tracking (see Sec. *11.7*).

Note: A simulation step usually executes one agent activity or signal handler. The first approach uses mobile notification agents to distribute sensor data to neighbour nodes (created on each event), which causes an up to 40 times higher communication cost compared with the other approaches. In the second approach distributed notification agents are sent to neighbour nodes one time, and using A2A signals to update sensor data on neighbour nodes managed by the notification agents. The third approach uses remote tuple access based on A2N signals, which causes the lowest communication cost and do not require previous agent distribution.

3.5 Process Communication Calculus

The π-Calculus introduced by Milner (1992) and the extended asynchronous distributed π-Calculus introduced by Hennessy [HEN07] (aDΠ) are common formal languages for concurrent and distributed systems, suitable for studying the behaviour and reaction of distributed and concurrent systems including dynamic changes caused by mobility. In the following subsection the relationship of the *AAPL*/DATG behaviour and interaction model with the Π-Calculus is pointed out, moving the view of point from spatially located agents in a distributed interconnected system to one unified concurrent system with dynamic virtual communication channels. The Π-Calculus used here extends the π-Calculus with the concept of (structural) domains, locations, resources associated with domains and locations, and migration of processes, close to the MAS paradigm, introduced in Section *2.8*.

3.5.1 The Π-Calculus and Tuple-Spaces

The reduction of the tuple-space communication to the Π-Calculus using communication channels is a non-trivial task. Since in this work communication is basically channel-based, rather than performed with distributed data spaces, formal approaches like, for example, *tKLAIM* [NIC07], modelling tuple-space interaction on an abstract level with (shared) data-based communication are not very well suited in a distributed system. The approach presented in [BRA05] that extends the π-Calculus with tuple-space operations and semantics is better suited as a starting point (*LinCa*, formal language).

A communication channel in the π-Calculus is shared by different agents (processes), the producer and consumer of tuples, and the tuple-spaces server, here represented by an agent-process, too. A polyadic channel has a type signature $\langle T_1, T_2, .. \rangle$ and is characterized by a fixed arity. Tuples only contain actual parameters, but template patterns contain formal and actual parameters, which must be handled differently in a channel-based interaction.

In a distributed system a tuple-space *TS* is bound to a specific location *l* and/or domain *d*. A first simplification is achieved by the decomposition of each tuple-space *TS* in sub-spaces sorting tuples by their arity (number of fields), i.e., TS_1, TS_2, .. , shown in Figure 3.3. This approach is also used for all agent processing platforms proposed in this work. This can be done due to the fact that tuples of different arities are always independent.

In the following let *TS* be a tuple space, $e = \langle d_1; d_2; \ldots ; d_n \rangle \in TS$ is a tuple of this space consisting of values d_i, and $t = \langle dt_1; dt_2; \ldots ; dt_m \rangle \in TP$ is a template pattern, which is request specific. Each tuple field element d_i is always a value *v*, which can be calculated by an expression (containing variables). Each template field element dt_i is either a value (actual parameter *v*) or a variable reference (formal parameter *p*). The type signature of a tuple is a type tuple of the form $Sig(e) = S_e = \langle T_1; T_2; \ldots ; T_n \rangle$. The type signature of a template pattern must be extended with the parameter kind tag pk=$\{t_v, t_p\}$, i.e., $Sig(t) = S_t = \langle T_1{:}pk_1; T_2{:}pk_2; \ldots ; T_m{:}pk_m \rangle$. The type signature of a tuple leads to a further decomposition of each n-ary subspace TS_n in a set of subspaces $\{TS_{n,S1}, TS_{n,S2}, ..\}$, differing in the type signature S_i of the tuples.

Keeping the partitioning of a tuple-space in n-ary sub-spaces *ts* in mind, all processes storing tuples with a specific arity using the output operation can use a shared communication channel. The tagged template type signature prevents using a shared communication channel for the reading of tuples based on templates pattern matching. More precisely, two different channels are required, one forwarding the actual parameters of the template to the TD_i server process, and one returning the values assigned to the formal parameters of the template. Furthermore, such a channel pair is request specific, and the channel pair must be created each time when data from the tuple-spaces is requested. Figure 3.3 illustrates this effect for a tuple-space supporting three-ary tuples.

In the following the modelling of the tuple-space operations, related to the *AAPL* model that was introduced in Section 2.9, using the Π-calculus algebra is shown, which was already introduced in Section 2.8.

Assume two agent processes PA_1 and PA_2 that exchange data by communicating with a tuple-space server process PT_n associated with a tuple-space containing tuples having the signature $Sig(ts_n)=\langle T_1; T_2; \ldots ; T_n \rangle$. Definition 3.2 shows the process notation for the tuple-space server. To satisfy the producer-consumer synchronization and data consistency of the tuple-space only one service request may be serviced at any time. This is ensured by the sequential functional (`rec self • (..); self`). All tuples of a specific tuple-space (the tuple set) are stored in a data list and modified by common list operations (nil: $() \rightarrow L$, $cons$: $E \times L \rightarrow L$, $head$: $L \rightarrow E$, $tail$: $L \rightarrow L$, $concat$:$L \times L \rightarrow L$).

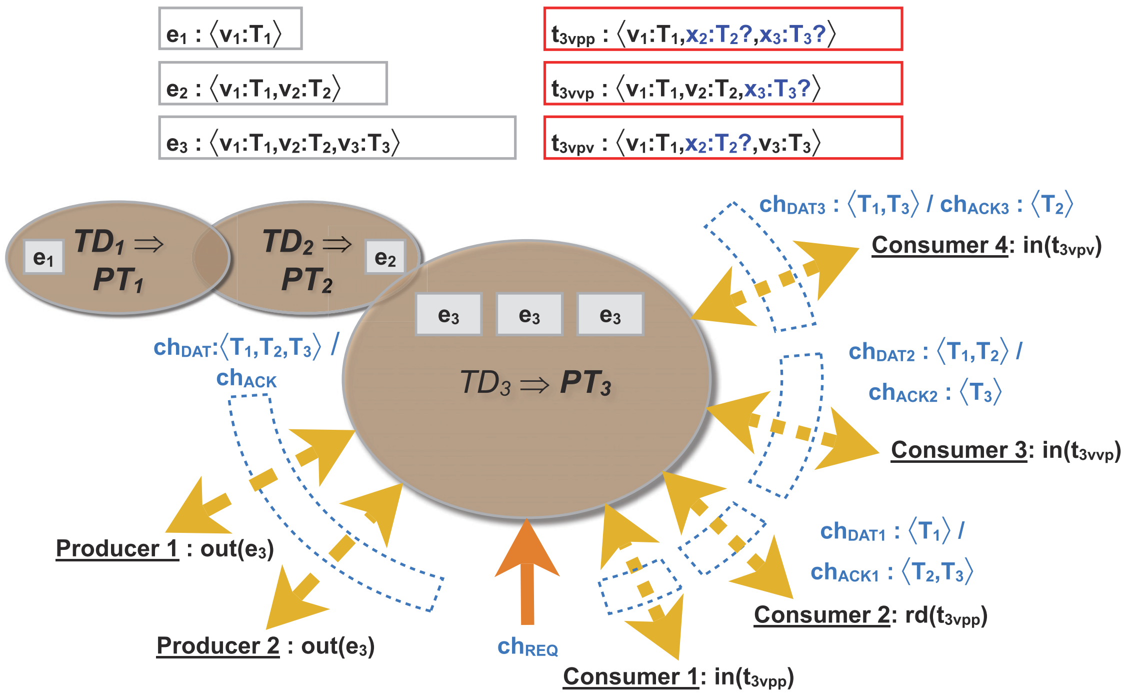

Fig. 3.3 *The decomposition of tuple-spaces in subspaces (tuple domain TD_i), each managing tuples of a specific arity (i) and type signature. There are different communication channels (CH_1,CH_2,..), one common for a specific tuple used by producer, and multiple private channel pairs for different template signatures used by consumer processes.*

The ordering of lists is pointless for tuple-spaces. The server process binds the data list variable `el` for storing tuples (in contrast, for example, in the Kernel Language for Agents Interaction and Mobility *KLAIM* treating tuples as processes). A server process has a global request channel $\texttt{req}_\texttt{i}$ bound to a location l_j.

A client (producer or consumer of tuples) sends a request over this channel. The request channel is polyadic with four element fields: the operation type from the set {*IN*, *OUT*, *RD*,..}, a tuple or template signature (specifying type and kind of parameters), the data channel communicating the tuple or the actual parameter of a template of the request, and the acknowledge channel used to reply to the client process.

An output request receives the tuple from the data channel, stores the tuple in the subspace list, and finally acknowledges the request, causing the producer process to proceed (i.e., a process reduction occurs).

An input request requires the splitting of the template pattern in the actual and formal parameters. Therefore, the request must be processed accordingly to the template signature, which is performed by matching the actual signature type with possible type patterns, e.g., as shown in Figure 3.3. The data channel only transmits the actual values of a pattern, finally expanded to a search template pattern by using the expand function and the provided signature *sig*, replacing all formal parameters with wild-card symbols. The acknowledge sent back to the consumer processes contains a collapsed tuple consisting only of values for the formal parameters of the template pattern, retrieved by the `collapse` function using one found tuple and the request type signature. All tuples matching the search template pattern are filtered by the `filter` function. If there was no matching tuple found, the request is sent back (queued loop-back) to the server request channel (in the Π-callous the server process can't control the consumer process directly, only by preventing the acknowledge message).

Def. 3.2 *Server process at location l_i servicing the tuple-space TS_i (in Π notation using channels)*

```
(new req_i@l_j)
Serv_i(req_i)@l_j ⇐
  (new el:⟨T_1; T_2; ... ; T_n⟩ list)
  rec self •
    (new op, sig, data, ack)
    req_i?(op, sig, data, ack) .
    ( match req with
      OUT:
        (new e) data?(e) . el ← cons(e,el); ack!()
      IN,RD:
        match sig with
        sig1:
          (new t_1:⟨T_1⟩) data?(t_1) .
          (new s) s ← filter(el,expand(t_1,sig))
            if s ≠ nil then (
            (new x) x ← head(s)
            if req = IN then (el←remove(el,x) )
           ack!<collapse(x,sig)>
          ) else ( req_i!<op,sig,data,ack> )
        sig2:
          (new t_2:⟨T_1; T_2⟩) data?(t_2) .
          (new s) s ← filter(el,expand(t_2,sig))
            if s ≠ nil then (
            (new x) x ← head(s)
            if req = IN then (el←remove(el,x) )
           ack!<collapse(x,sig)>
          ) else ( req_i!<op,sig,data,ack> )
        ..
     ); self
```

The client process notation and transformation for the output and input tuple-space operations and the usage of the three communication channels involved in each request is shown in Equations 3.1 and 3.2.

$$\frac{\mathrm{out}(v_1,v_2,..)@l}{(\mathrm{new}\ ch_d, ch_a)\ (req_i@l!\langle \mathrm{OUT}, ch_d, ch_a\rangle\ .\ ch_d!\langle v_1,v_2,..\rangle) \parallel ch_a?()\ .\ \mathrm{P}} \tag{3.1}$$

$$\frac{in(t)@l \text{ with } t=(v_1,x_1?,v_2,x_2?,..)}{\begin{array}{l}(\mathrm{new}\ ch_d, ch_a)\\ (req_i@l!\langle \mathrm{IN},\mathrm{Sig}(t), ch_d, ch_a\rangle\ .\ ch_d!\langle v_1,v_2,..\rangle) \parallel\\ ch_a?(x_1,x_2,..)\ .\ \mathrm{P}\end{array}} \tag{3.2}$$

A tuple-space is bound to a specific location, here the network node. Two processes PA_1 and PA_2 can only synchronize and exchange a tuple (assuming *match*(*e*,*t*)=*true*) if they access the same tuple-space *TS@l* at the same location *l*, shown by the following reduction rules.

$$\frac{\mathrm{out}(e_1)@l.P@l \parallel \mathrm{in}(t_1)@l.Q@l}{P \parallel Q} \qquad \frac{\mathrm{out}(e_1)@l_1.P@l \parallel \mathrm{in}(t_1)@l_2.Q@l_2}{P@l_1 \parallel \mathrm{in}(t_1)@l_2.Q@l_2} \tag{3.3}$$

Up to here it is assumed that producer and consumer processes (P_p, P_c) exist during the exchange of tuples. But tuples are persistent and not requiring the existence of the producer process after it has generated a tuple.

The lifetime of the tuple *T* is independent of the lifetime of the producer process P_p. For this reason, the tuple exchange requires the tuple-space server process P_s and tuples cannot be exchanged directly by producer and consumer processes using channels if the producer process does not exist, though this approach would be natural and a significant simplification. Also, if tuples are represented by communicating processes themselves (as proposed in [NIC96] and [NIC98]), producer-tuple and tuple-consumer communication cannot be performed by using channels without the presence of a coordination server if the consumer request happened before the producer generates a matching tuple. The different cases with and without a server are illustrated in Figure 3.4.

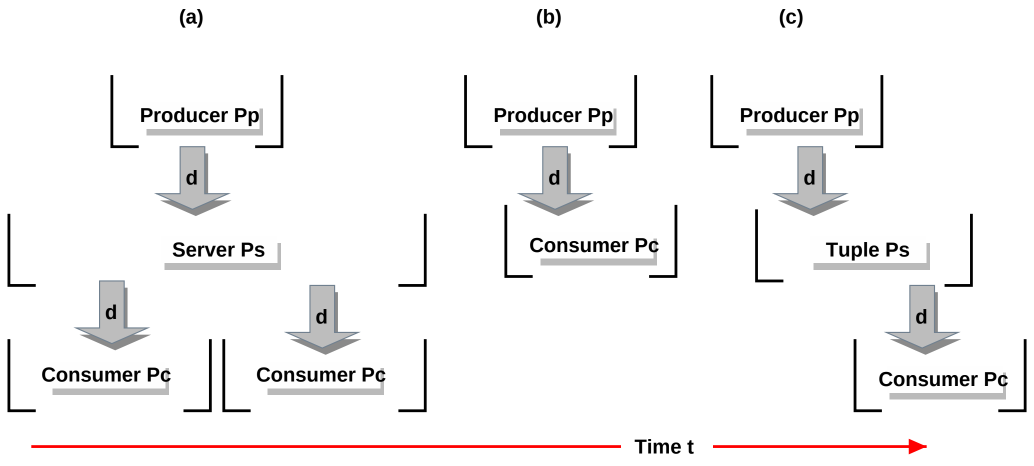

Fig. 3.4 *Different producer-consumer situations with and without a coordinating tuple-space server [(a) with server, (b) process-to-process, (c) tuples as processes).*

Another issue arise with tuple exchanges without a coordination server if there are multiple originally independent tuples matches an input request, which cannot be handled in the cases (b) and (c) in Figure 3.4. At least the coupling of tuple processes is required to coordinate template matching and scheduling.

3.5.2 The Π-Calculus and Signals

The *AAPL* signals can be sent from one agent Ag_n to another Ag_m (peer-to-peer, peer-to-group, or bounded broadcast). A signal can be considered as being a communication channel connecting two agent processes. A signal handler of an agent is related to a separate process operating concurrently to the parent agent process. But there is no direct synchronization between the parent and signal handler process. Only the shared agent body variables are used to communicate data, basically not part of the CSP and Π-Calculus. Usually an ATG transition is blocked until a Boolean condition expression containing body variables is true. This transition blocking can be modelled with a channel communication, too, as well as the signal propagation itself. The mapping of *AAPL* signal propagation and handling on the Π-calculus is shown in Definition 3.3.

Def. 3.3 *Mapping of AAPL signal propagation and handler on the Π-calculus (c: conditional Boolean expression)*

AAPL

```
Agt_a:
              A_i → A_j: c(v_1,v_2,..,v_n)
              handler S(arg) = .. v_i ← ε; ..

Agt_b:
              send(Agt_i,S,va)
```

Π-Calculus

$P_{a,i,wait} \Leftarrow \text{rec } p \bullet \text{if not } c(..) \text{ then } (\underline{event_a}?() \;.\; p)$

$P_{a,i} \Leftarrow P_{a,i,comp1} \cdot P_{a,i,wait} \cdot P_{a,i,trans,j}$

$H_a \Leftarrow \text{rec } h \bullet \underline{S_a}?(arg) \;.\; v_i \leftarrow \varepsilon;\ \underline{event_a}!();\ h$

$P_b \Leftarrow \underline{S_a}!\langle arg\rangle$

3.6 FIPA ACL

Agent communication can be achieved basically by three different methods: 1. Signal propagation (similar to commitment messages in *AGENT0*, [SHO91]), 2. Tuple database exchange, and 3. Using agents with a composition of methods 1 & 2. These basic methods can be used to realize common higher-level agent communication languages like *ACL* or *KQML* (tuple patterns correspond to message content entries). Signal propagation implements light-weighted asynchronous peer-to-peer Remote-Procedure calls, executed on target agents with appropriate signal handlers, which must not necessarily belong to the same agent class, whereas pattern matching based tuple database access can be performed by any group of agents having a common understanding of the meaning of data and which are actually processed on the same platform node.

For example, a simple FIPA *ACL* based request from agent *A* (initiator) to *B* (participant), which ask for a database tuple on *B* can be created with the following *AAPL* code pattern using signals, shown in Example 3.2.

Ex. 3.2 *FIPA ACL and AAPL*

FIPA ACL

```
(request :sender IDA
  :receiver IDB :content (p ?) :ontology TS2)
```

AAPL

```
ξ: {REQ1,REQ2,INFR,FAIL}
```

```
          χ: {COMERR}

Agent A
          Σ: {pv,ps}
          ξ INFR: v → {
            pv ← v;
            ps ← true;
            wakeup
          }

          ξ FAIL: {
            ps ← false;
            wakeup
          }

          request: (AID,p) → {
            ξREQ1 (p) ⇒ AID; τ+(100,FAIL);
            sleep
            if ps then pv else ↑COMERR end
          }

Agent B
          ξ REQ1: arg →
            σ: {v}
            if ∇?(p,?) then
              ∇-(p,?v)
              ξINFR (v) ⇒ $sender
            else ξFAIL ⇒ $sender ...
```

3.7 AAPL Agents and Capability-based Remote Procedure Calls

One common inter-process communication model is the Remote-Procedure call paradigm, performed by clients and servers, though the roles can be fully interchanged, i.e., a server can be a client if the server requests a procedure execution on another server.

Considering the well-known capability-based Amoeba RPC interface [MUL90], shown in Definition *3.4*, this procedural RPC programming interface can be easily mapped on the *AAPL* agent model, discussed later.

Def. 3.4 *Simplified and abstracted Amoeba RPC interface using capabilities [ai: Argument, xi?: parameter replaced with argument values]*

```
Capability
          type capability = (port:bytes[6],obj:bytes[4],
                              rights:byte,priv:byte[6])
```

```
Client
          stat := trans(cap,a1,a2,..,x1?,x2?,x3?,..)
Server
          loop
            getreq(port,a1?,a2?,...)
            compute
            putrep(port,x1,x2,..)
```

The server resources are specified with capabilities. A capability therefore holds the server port and additional information about the object. The entries have the following meaning:

port (P1..P6)
The public server port (6 bytes),

obj (O)
The object number given by the server (4 bytes). It is a unique server internal identification number of this object,

rights (R)
The rights mask (1 byte) determines the allowed access rights of an object, like the permission to destroy an object. Each bit in the rights field specifies one possible access right. The meaning of each bit is dependent of the server and the kind of the object.

priv (S1..S6)
The security private port (6 bytes). This port protects the rights field against manipulation.

The rights protection port contains the rights field that is created by a one-way encoding function *fc* from a private check port *C* randomly created by the server only for this object and the rights field. A restricted capability *CAP'* is build from an original one by restricting the rights field and creating a new security port using an encode function *pc*. This function simply calculates the new security port S' from the private check port *C* and the rights field *R* using a logical *XOR* operation and feeds the result to a one-way function *Fc*, as shown in Equation 3.4. Each time a server receives a message it checks the security port using his private check port *C* and a decode function *pd*. This function simply builds the expected security port *S'* from the rights field specified in the received capability and the check port *C* and compares S' with the supplied *S* port. If they are unequal, the capability was manipulated and will be rejected.

$$S' = Fc(S \text{ xor } R) \tag{3.4}$$

A server uses a `getreq` function to wait for the arrival of client request matching the server port. A client transaction using the `trans` function is synchronous and waits for the reply, send by the server using the `putrep` function (see Figure 3.5, left side).

DNS: Capability-Name Mapping. Capabilities are not initially associated with a name. A hierarchical directory based approach can be used to associate a set of capability {$C_1(P,O,R_1,S_1)$, $C_1(P,O,R_2,S_2)$, ..} with a set of names N={n_1,n_2,..}, referencing the same object and server. A directory is a (*name*,*capability*) table with its own object capability that is managed by a Directory Name Server (*DNS*, see Figure 3.5, right side).

This RPC interface can be implemented with mobile client and server agents as subclasses that are forked from a root agent class, implementing some specific agent behaviour, shown in Algorithm 3.1. The RPC extension allows different forked child agents to execute activities of other child agents (the servers), which may me mobile, too. If the server and clients belong to the same root agent (children), the above security handling can be omitted, and the client-server transaction can always be trusted.

Each time an agent (of the root class *ac*) wants to execute a RPC on a remote (or local) agent, it creates (forks) a transaction agent. The transaction agent inherits the current set of arguments {a_1, a_2,..} and the target server port capability *cap*. Before the transaction agent can interact with the server agent, it must locate the server owning the server port *cap.port*.

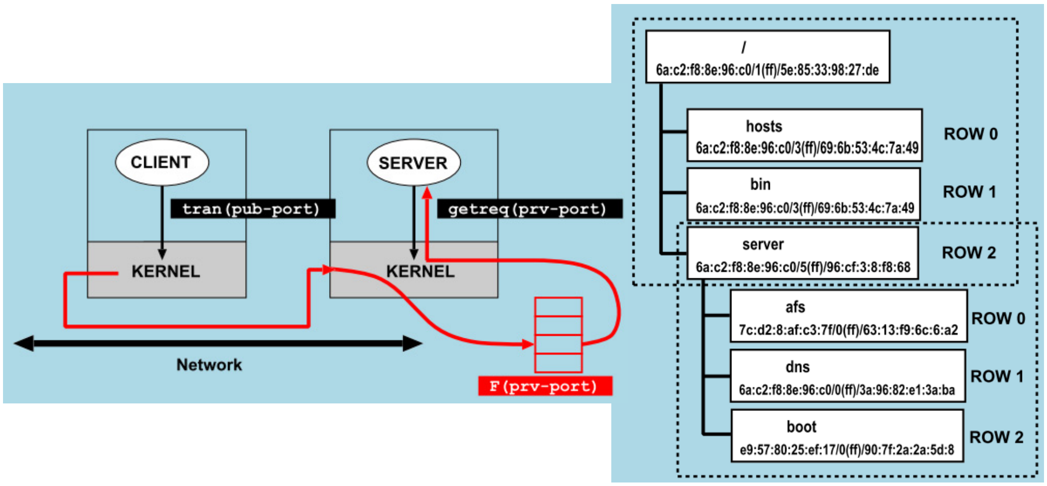
CLIENT
tran(pub-port)
KERNEL
SERVER
getreq(prv-port)
KERNEL
Network
F(prv-port)
/
6a:c2:f8:8e:96:c0/1(ff)/5e:85:33:98:27:de
hosts
6a:c2:f8:8e:96:c0/3(ff)/69:6b:53:4c:7a:49
ROW 0
bin
6a:c2:f8:8e:96:c0/3(ff)/69:6b:53:4c:7a:49
ROW 1
server
6a:c2:f8:8e:96:c0/5(ff)/96:cf:3:8:f8:68
ROW 2
afs
7c:d2:8:af:c3:7f/0(ff)/63:13:f9:6c:6:a2
ROW 0
dns
6a:c2:f8:8e:96:c0/0(ff)/3a:96:82:e1:3a:ba
ROW 1
boot
e9:57:80:25:ef:17/0(ff)/90:7f:2a:2a:5d:8
ROW 2

Fig. 3.5 *Amoeba RPC (left) and Directory Capability Name Service (right)*

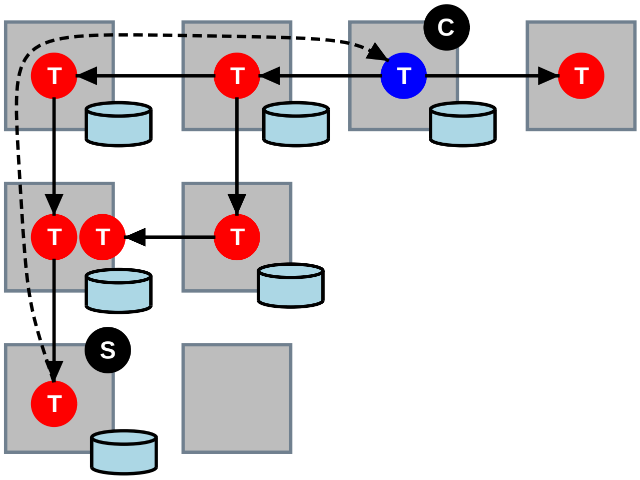

Fig. 3.6 *RPC server search with a distributed MAS [C: Client agent, T: Transaction agent (red: search agent), S: Server agent]*

The search of the server is performed with a divide-and-conquer approach, similar to the explorer SoS that will be introduced in Section 9.2, shown in Figure 3.6.

The server search is performed in the *lookup* activity. If the server is not existing on the current node, forked transaction search agents will explore the neighbourhood, and in the case one agent found the server port, it will deliver the relative distance information back to the waiting transaction agent, performed by the *iamhere* activity.

The location information (*dx*0, *dy*0) will be published in the local tuple database (acting as a cache) and during back propagation on each node along the back path.

The original transaction agent forks and sent out the child search agents in all reachable directions in the *lookup* activity. Each child agent moves to the specified neighbour node. The created child search agents perform a transition to the *iamhere* activity if they found the server. If the server was not found, it will fork and sent out more agents in all directions excluding its own forward and backward direction. If a search agent finds the server, it will go back to its origin and notifies the original transaction agent (*iamhere* activity).

If the search of the server location was successful, the transaction migrates to the server place and performs the transaction. To ensure that the server agent is not migrated in the meantime, the server port is checked again. If the server is gone, the transaction fails (and the agent dies) After the transaction is finished, it moves back to the root place and delivers the data to the parent agent though the tuple-space.

The previously introduced RPC client-server agent subclasses as part of an application agent class can alternatively be embedded in their own RPC root class.

Alg. 3.1 *AAPL RPC Implementation with mobile client and non-mobile server agents, both embedded as subclasses in a root class*

```
Ψ someagentclass: (dir,radius) → {
  Cap: (port:natural,obj:natural)
  Π : {cap,req,a1,a2,..,x1,x2,...
  ξ: {IAMHERE}
  ..
  inbound: (nextdir) → {
    case nextdir of
    | NORTH →  dy > -radius
    | SOUTH →  dy < radius
    | WEST  →  dx > -radius
    | EAST  →  dx < radius
  }

  α init:{
    port ← ℜ{1..nmax} must be public in the root class
  }

  α client: {
   req <- ℜ{1..nmax}
    out(TRANS1,cap.port,cap.obj,req,a1,a2,..) superfluous,
                                  forked agent inherits all arguments
    Θ⁺ac.trans(ORIGIN,4)
    stat ← ∇⁻?(1 sec,TRANS2,cap.port,req,x1?,x2?,..)
  }

  φ trans: {
    ↓ a1,a2,..,x1,x2,...
    Π: {transid,dx,dy,dx0,dy0,cap:Cap}
    π: {port,req,obj,priv,found}

    α init: {
     found ← false
     transid ← $self
      ∇⁻(TRAN1,port,obj,req,a1?,a2?,..) superfluous, forked agent
                                        inherits all arguments
    }

    α lookup: {
     check for desired server port
      if ∇?(CAP,cap.port,?,?) then
```

```
        ∇%(CAP,cap.port,dx0?,dy0?)
       if (dx,dy) ≠ (0,0) then
        remote lookup, deliver port location to waiting trans agent
         incr(dx0,dx);
         incr(dy0,dy)
         π*(lookup,iamhere)
       else
         client and server have same location, start transition..
          found ← true
     else
       if (dx,dy) = (0,0) then
         this is the initial transtion agent, fork search agents..
          π*(lookup,move)
          ∀{nextdir ∈ ω | ?Λ(nextdir)} do
            Θ→ac.trans(nextdir,radius)
          π*(lookup,move,found=true)
          π*(move,putreq)
       else
         this is already a search agent, fork more search agents..
         ∀{nextdir ∈ ω | nextdir≠dir &
                         nextdir≠ϖ(dir) &
                         ?Λ(nextdir)} do
            Θ→ac.trans(nextdir,radius)

    }

    α iamhere: {
      ∇+(CAP,cap.port,dx0-dx,dy0-dy)
      | if dy > 0 then decr(dy); ⇔(SOUTH)
      | if dy < 0 then incr(dy); ⇔(NORTH)
      | if dx > 0 then decr(dx); ⇔(WEST)
      | if dx < 0 then incr(dx); ⇔(EAST)
      | if (dx,dy) = (0,0) then ξIMAHRE ⇒ transid
    }

    α putreq: {
      if ∇?(CAP,cap.port,0,0) then
        ∇+(CAP,cap.port,req,cap.obj,a1,a2,..)
      else Θ×($self)   destroy this agent, server is gone
    }

    α getrep: {
      ∇-(REP,cap.port,req,,x1?,x2?,..)
    }

    α move: {
      if inbound(dir) then
```

```
          (dx,dy) ← (dx,dy) + δ(dir)
          ⇔(dir)
        else Θˣ($self)   destroy this agent, server not found
      }

      α deliver: {
        | if dy > 0 then decr(dy); ⇔(SOUTH)
        | if dy < 0 then incr(dy); ⇔(NORTH)
        | if dx > 0 then decr(dx); ⇔(WEST)
        | if dx < 0 then incr(dx); ⇔(EAST)
        | if (dx,dy) = (0,0) then
           ∇⁺(TRANS2,cap.port,req,x1,x2,..)
           Θˣ($self)
      }

      ξ IMAHRE: {
        found ← true
      }

      π: {
       init → lookup
        lookup → move | found
        imahere → iamhere | (dx,dy) ≠ (0,0)
        getrep → deliver
        deliver → deliver | (dx,dy) ≠ (0,0)
      }
    }

    φ server: {
      α init: {
        ∇⁺(CAP,port,0,0)
      }
      α getreq: {
        ∇⁻(REQ,port,req?,obj?,a1,a2,..)
      }
      α service: {
        compute f: (a1,a2,..) → (x1,x2,..)
      }
      α putrep: {
        ∇⁺(REP,port,req,x1,x2,..)
      }
      π: {
        getreq → service
        service → putrep
        putrep → getreq
      }
    }
```

One disadvantage of tuple spaces introduced so far is related to the flat organisation model of a tuple space, compared, for example, with tree based directory organisation providing hierarchy and path references of objects (data or directories). Actually tuples are only matched by their dimension and template equivalence.

The capability concept can be extended to tuple spaces in such a way that a tuple space and a tuple server is associated with a capability. An eventually distributed (replicated) directory service organizes tuple spaces hierarchical with trees. Nodes offering tuple spaces can publish the capability in a directory, which can now be referenced by a textual path, easing the selection of tuple spaces. This extends the tuple space operations *in/out/rd* (..) with an extra argument, the server capability.

3.8 Further Reading

1. R. Milner, The space and motion of communicating agents. Cambridge University Press, 2009.
2. M. Hennessy, A Distributed PI-Calculus. Cambridge University Press, 2007.
3. R. Bordini, J. Hübner, and M. Wooldridge, Programming multi-agent systems in AgentSpeak using Jason. 2007.
4. M. Schumacher, Objective Coordination in Multi-Agent System Engineering. Springer Berlin, 2001.