Final draft: Karel Kruger & Anton Basson (2016): Erlang-based control implementation for a holonic manufacturing cell, International Journal of Computer Integrated Manufacturing,
DOI: 10.1080/0951192X.2016.1195923

Erlang-based Control Implementation for a Holonic Manufacturing Cell

Karel Kruger [a] and Anton Basson [a,*]

[a] Dept of Mechanical and Mechatronic Eng, Stellenbosch Univ, Stellenbosch 7600, South Africa
[*] Corresponding author. Tel: +27 21 808 4250; Email: ahb@sun.ac.za

Abstract

Holonic control is generally used in reconfigurable manufacturing systems since the modularity of holonic control holds the promise of easier reconfiguration, reduction in complexity and cost, along with increased maintainability and reliability. As an alternative to the commonly used agent-based approach, this paper presents an Erlang-based holon internal architecture and implementation methodology that exploits Erlang's capabilities. The paper shows that Erlang is well suited to the requirements of holonic and reconfigurable systems - due to strong modularity, scalability, customizability, maintainability and robustness characteristics.

Keywords: Erlang/OTP, Holonic manufacturing system (HMS), Reconfigurable manufacturing system (RMS)

1 Introduction

The concept of Reconfigurable Manufacturing Systems (RMSs) is aimed at addressing the needs of modern manufacturing, as have been shaped by aggressive global competition and uncertainty resulting from dynamic changes in economical, technological and customer trends (Leitao and Restivo 2006). The critical requirements for modern manufacturing systems include (Bi et al. 2008) short lead times for the introduction of new products into the system, the ability to produce a larger number of product variants and the ability to handle fluctuating production volumes.

RMSs aim to switch between members of a particular family of products, by adding or removing functional elements (hardware or software), with minimal delay and effort (Vyatkin 2007). RMSs are also designed to be able to rapidly adjust the production capacity and functionality in response to sudden changes, by reconfiguring hardware and control resources (Bi et al. 2008; Bi, Wang, and Lang 2007). RMSs therefore should be characterised by (Mehrabi, Ulsoy, and Koren 2000; ElMaraghy 2006): modularity of system components, integratability with other technologies, convertibility to other products, diagnosibility of system errors, customizability for specific applications and scalability of system capacity..

A popular approach for enabling control reconfiguration in RMSs is holonic control architectures. The term holon (first introduced by Koestler in 1967) comes from the Greek words "holos" (meaning "the whole") and "on" (meaning "the particle"). Holons are then "any component of a complex system that, even when contributing to the function of the system as a whole, demonstrates autonomous, stable and self-contained behaviour or function" (Paolucci and Sacile 2005). When this concept is applied to manufacturing or assembly systems, a holon is an autonomous and cooperative building block for transforming, transporting, storing or validating information of physical objects. A Holonic Manufacturing System (HMS) is then "a holarchy (a system of holons which can cooperate to achieve a

common goal) which integrates the entire range of manufacturing activities" (Paolucci and Sacile 2005).

The application of the holonic concept to manufacturing control systems has been a popular field of research since the early 1990's. Even though several experimental implementations have been reported, predominantly based on agent based programming (such as Leitao and Restivo [2006]]), we believe there is room for improvement in terms of reduced complexity, greater potential for industry acceptance, better robustness/fault-tolerance and better inherent scalability.

This paper evaluates a new alternative to agent based approaches: the implementation of holonic control using the Erlang programming language. Erlang is a concurrent, functional programming language which was developed for programming concurrent, scalable and distributed systems. In Erlang, many lightweight processes can be employed to work concurrently while distributed over many devices. Processes are strongly isolated, having no shared memory, and can only interact through the asynchronous sending and receiving of messages (Armstrong 2003). The Erlang programming environment is supplemented by the Open Telecommunications Platform (OTP) - a set of robust Erlang libraries and design principles providing middle-ware to develop Erlang systems (Anonymous s.a. [a]; Logan, Merrit, and Carlsson 2011).

The objective of this paper is to present an Erlang-based internal architecture for holons and an implementation methodology, targeting a reconfigurable manufacturing system. A resource holon in the PROSA holonic control architecture (discussed in section 2.2) is used as a prototype since it contains all the architectural elements required for the other holon types, as well as hardware interfacing.

2 Holonic Control

This section motivates the use of the holonic control approach and gives some background regarding reference architectures. The generic resource holon model, used for the Erlang implementation, is also discussed.

2.1 Advantages of Holonic Control

The use of holonic control for RMSs holds many advantages: Holonic systems are resilient to disturbances and adaptable in response to faults (Vyatkin 2007); have the ability to organise production activities in a way that they meet the requirements of scalability, robustness and fault-tolerance (Kotak et al. 2003); and lead to reduced system complexity, reduced software development costs and improved maintainability and reliability (Scholz-Reiter and Freitag 2007).

2.2 Holonic Architecture

The full utilization of the above-mentioned advantages relies on the holonic system's architecture. Several reference architectures, which specify the mapping of manufacturing resources to holons and to structure the holarchy, have been proposed (e.g. Chirn and McFarlane [2000]; Leitao and Restivo [2006]), but the most prominent is PROSA (Product-Resource-Order-Staff Architecture) (Van Brussel et al. 1998).

PROSA defines four holon classes: product, resource, order and staff. The first three classes of holons can be classified as basic holons, because, respectively, they represent three independent manufacturing concerns: product-related technological aspects (product holons), resource aspects (resource holons) and logistical aspects (order holons).

The basic holons can interact with each other by means of knowledge exchange, as is shown in Figure 1. The process knowledge, which is exchanged between the product and resource holons, is the information and methods describing how a certain process can be achieved through a certain resource. The production knowledge is the information concerning the production of a certain product by using certain resources – this knowledge is exchanged between the order and product holons. The order and resource holons exchange process execution knowledge, which is the information regarding the progress of executing processes on resources.

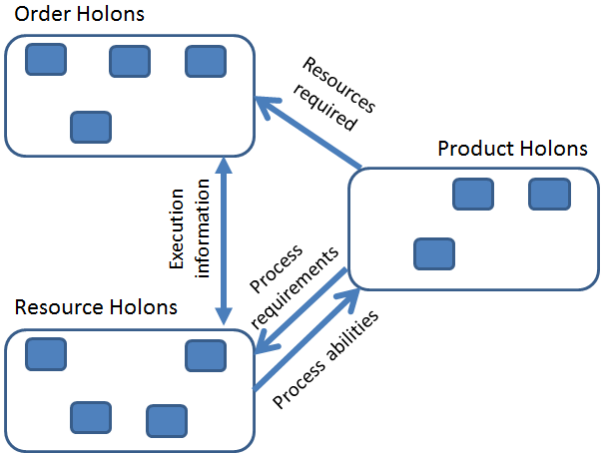

Figure 1. Basic Holons of PROSA (Van Brussel et al 1998).

Staff holons are considered to be special holons as they are added to the holarchy to operate in an advisory role to basic holons. The addition of staff holons aim to reduce work load and decision complexity for basic holons, by providing them with expert knowledge.

The holonic characteristics of PROSA contribute to the different aspects of reconfigurability mentioned in section 1. The ability to decouple the control algorithm from the system structure, and the logistical aspects from the technical aspects, aids integrability and modularity. Modularity is also provided by the similarity that is shared by holons of the same type.

2.3 Resource Holon Model

The paper uses the resource holon as case study because of the range of capabilities that is required, such as communication, execution control and hardware interfacing. The resource holon model used as starting point is described in this section – an adapted model for implementation with Erlang follows in section 0.

The internal architecture of a resource holon is illustrated in Figure 2. Individual holons have at least two basic parts (Kotak et al. 2003; Leitao and Restivo 2002): a functional component and a communication and cooperation component. The functional component can be represented by a purely software entity or, as in resource holons, it could be a hardware interface represented by a software entity. The communication and cooperation component of a holon is implemented by software.

The communication component is responsible for the inter-holon information exchange. The decision-making component is responsible for the manufacturing control functions, regulating the behaviour and activities of the holon. The interfacing component handles the

intra-holon interaction, providing mechanisms to access the manufacturing resources, monitor resource data and execute commands in the resource.

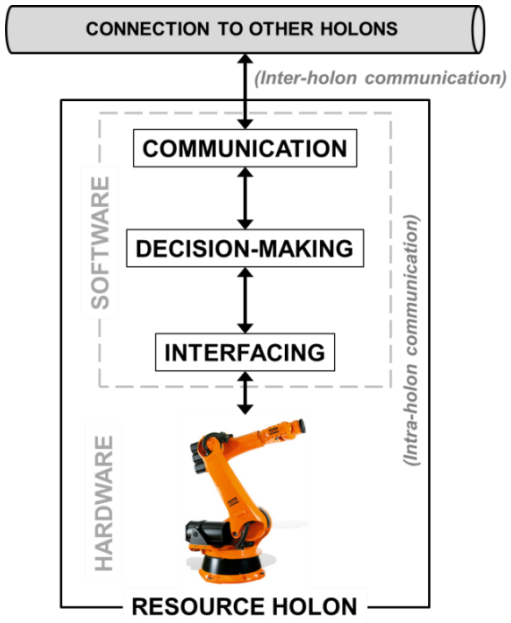

Figure 2. Internal architecture of a resource holon (adapted from Leitao and Restivo 2006).

3 Advantages of using Erlang for Holonic Control Implementation

There are several inherent characteristics of Erlang which prove to be advantageous for the implementation of holonic control. The most prominent advantages relate to fault-tolerance, service availability and scalability.

The Erlang process model – whereby system functionality is distributed across a number of cooperating and communicating processes – ensures that Erlang is built on an inherently fault-isolating architecture. The processes act as abstraction boundaries, limiting the propagation of error through the system (Armstrong 2003). This strong fault-tolerant nature of Erlang is further supplemented by the OTP libraries for supervisory structures, which can be utilized to detect and trap system errors and implement strategies to rectify the system behaviour (Armstrong 2003).

Erlang allows for the updating of code without having to disturb the operation of a running program since it has primitives which allow code to be replaced in a running system (Däcker 2000). Bug fixes and upgrades can be uploaded to a running system without disturbing the current operation. This capability, along with the previously mentioned fault-tolerance, enables Erlang systems to offer excellent service availability (Armstrong 2007).

Finally, Erlang provides the infrastructure for massive scalability and concurrency. The light-weight nature of Erlang processes means that millions of processes can be supported on a single processor (Armstrong 2007). Furthermore, since Erlang processes share no memory and all interaction is done through message passing, processes can easily be distributed over a network of processors (Armstrong 2003).

A comprehensive comparison of Erlang with other implementation options is beyond the scope of this paper. However, from the authors' experience, the following comments are offered:

Multi-agent systems (MASs) have been often been used to implement holonic control architectures for manufacturing stations and cells. Interestingly, the advantageous characteristics of Erlang can be directly related to what has been identified as the shortcomings of commonly used agent based implementations. Almeida et al. (2010) identified that two of the main issues regarding agent based implementations are that of scalability and fault-tolerance. Due to the high resource requirements of MAS threads (when implemented in Java or C [Vinoski 2007]), the number of threads that can run on a processor limits scalability – this limitation is emphasized when the implementation is to be done on resource-constrained industrial controllers. In terms of fault-tolerance, there is still work to be done on the implementation of supervisory structures which can identify, diagnose and recover from disturbances or errors.

When considering specifically the Java Agent DEvelopment (JADE) framework, which is often used for holonic control implementations, JADE agent threads suffer drawbacks concerning scalability, as mentioned above, since they Java based. Furthermore, JADE is aimed at providing infrastructure for a wider range of implementations (i.e. beyond that of control applications for manufacturing systems), but this infrastructure is mostly underutilized in the type of implementations presented in this paper. In some cases, this additional functionality adds complexity and coding overhead – a scenario where the sense of "scalable complexity" (the idea that a system can be constructed through the inclusion of only the functions and interfaces for the necessary functionality, and thus complexity, of the system) of Erlang implementations could be beneficial. Lastly, it has been found that programming MASs, even with Java programming experience, involves a significant learning

IEC 61131-3 languages are commonly used for control implementation in manufacturing. While they work well for low level control, attempts to use these languages for implementations of higher level control have achieved limited success. The reason for this, in the experience of the authors, is that the features of these languages that contribute to their reliability on the other hand restrict the flexibility and extensibility of the code that are valuable for the implementation of the high level control of holonic systems. Examples of these restrictions are that the programmes nominally operate in a single thread and that dynamic instantiation of objects, variables or data containers is not possible.

Object orientated programming (OOP) languages offer features between MASs and IEC 61131-3 languages, and can therefore also be considered for developing holonic control systems (Graefe and Basson 2013). C# and JAVA appears to have a wide user base in the software world, but their popularity in manufacturing control is uncertain. The authors' research group have found C# to be a productive tool to develop holonic control systems, utilising the classical OOP features. C# has the advantage above JAVA that drivers for I/O devices are more readily available for C#. However, the resource implications of multiple threads in C# are similar to that for JAVA. Also, neither of these languages include the "built-in" fault-tolerance and fault-management of Erlang.

4 Erlang-based Resource Holon

The internal holon architecture, inter- and intra-holon communication and the holon functional components are discussed in this section. Furthermore, a general implementation

methodology is described and an implementation case study for the Erlang-based resource holon is presented.

4.1 Internal Architecture

For the Erlang/OTP implementation, the internal architecture described in section 2.3 has been adapted to that shown in Figure 3. Though the *Communication* and *Interfacing* components are present in both models, the *Decision-making* component in Figure 2 is split into two components, namely the *Agenda Manager* and *Execution* components.

The division of the *Decision-making* component into the *Agenda Manager* and *Execution* components (discussed in sections 4.2.2.2 and 4.2.2.3) is motivated by two factors: Firstly, for a separation of functionality. By separating the functionality of handling service bookings and that directly concerning execution, reconfigurability is improved – the way in which bookings are handled and how a process must be executed can be changed independently and with minimal effect on the other component. Secondly, for software reusability: while the execution control may differ from holon to holon, the way in which their services are managed is similar. The *Agenda Manager* component can thus be used as a generic inclusion for every service-rendering holon in the system.

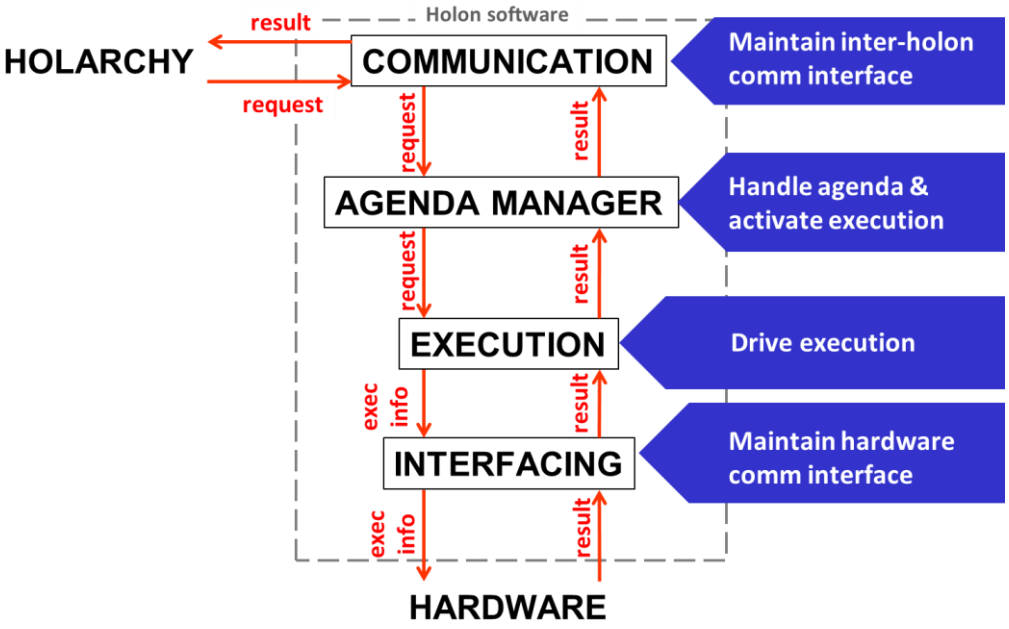

Figure 3. Resource holon model for the Erlang/OTP implementation.

4.2 Implementation Methodology

This section presents a general implementation methodology for a holonic control system with Erlang/OTP processes. A generic approach to facilitating communication and implementing the holon functional components is described.

4.2.1 Facilitating Communication

4.2.1.1 Inter- and Intra-Holon Communication

In holonic systems, communication between system entities can be classified as either inter- or intra-holon communication. Inter-holon communication refers to communication between different holons in the system, while intra-holon communication occurs between the internal components of a holon.

A typical example of inter-holon communication is the request of a resource holon service by an order holon – the order holon sends a request to the resource holon to which the resource holon replies with a request result. These *request* and *result* messages are shown in Figure 3 as interchanged by the *Holarchy* and the resource holon's *Communication* component. In addition to the inter-holon communication, Figure 3 also shows intra-holon communication - indicated as the exchange of *requests*, *results* and *execution information* between the functional components of the resource holon.

4.2.1.2 Messaging in Erlang

The Erlang process model dictates that information can only be shared amongst processes through messages. Messages are sent using the message operator "`!`" in the following format: `Receiver ! Message`. `Receiver` is a variable[1] that stores the process ID or registered name of the receiving process and the received message is stored in the `Message` variable. Messages can be received by using the `receive` statement with pattern matching, usually implemented in a loop (shown in section 4.3.1).

For increased traceability, the format by which messages are sent can be implemented as `Receiver ! {Sender,Message}`. In this case, the message payload is placed within a tuple together with the process ID or registered name of the process sending the message. This format offers more options on the receiving side, as pattern matching can then be performed on both the type and content of the message, and from where the message originated.

To further facilitate communication, an ontology can be incorporated in the implementation. The ontology definition can be done in a one or many separate header files, and included in the necessary modules. Using *records*, an Erlang data type similar to *structs* in C, sets of information can be defined and used in creating messages and matching messages to patterns. *Records* allow for data fields to be accessed by name instead of order, and multiple *records* can be nested to accommodate complex sets of information. An example of a *record* used to define service messages is shown in section 4.3.1.

4.2.1.3 Communication in Functional Components

Taking advantage of the light-weight nature of processes, leading to cheap and easily-managed concurrency, each functional component of the resource holon will be implemented as one or more Erlang processes. For the components to cooperate, information must be exchanged by means of messages. For this reason, each functional component must employ a process which handles this communication.

A simple way to facilitate the communication is to spawn a concurrent process running a *receive-evaluate* loop. The process calls a recursive function which implements a `receive` statement, followed by a set of patterns which will be matched against incoming messages. Upon successfully matching to a pattern, some action can be taken (usually the sending of another message). After each matching case, the function calls itself, resulting in a continuous loop.

The communication process described above separates the communication functionality, within a functional component, from the execution logic. This separation increases the reconfigurability and maintainability of the implementation, as changes can be made to one process without influencing the functionality of the other.

[1] Variables in Erlang start with a capital letter.

4.2.2 Implementing the Holon Functional Components

4.2.2.1 Communication Component

The *Communication* component of the resource holon is responsible for maintaining the communication interface with the rest of the holarchy – i.e. all messages to and from other holons are handled by this component.

This component can be implemented using only the communication process discussed in section 4.2.1.3. This process then allows for concurrency in the communication and execution functionality of the holon – i.e. the *Communication* component can operate uninterrupted and independent of the other functional components.

4.2.2.2 Agenda Manager Component

The agenda, in the context of this paper, refers to a list of service commitments (bookings) made by a resource holon to requesting order holons. The construction and management of such a list provides opportunity for the implementation of strategies to improve the performance of holonic systems by planning ahead through forecasting and tentatively committing future availability of resources. Two possible strategies that can be implemented are delegate multi-agent systems (D-MAS) (Holvoet and Valckenaers 2006) and a facilitating supervisor as found in ADACOR (Leitao and Restivo 2006). With D-MAS, holons delegate the responsibility of populating and consulting the agendas of resource holons to a swarm of light-weight agents. In ADACOR, a supervisor holon facilitates the booking of resource services by task holons, according to forecasts and optimized plans based on the inspection of agendas. Since the implementation of the mentioned strategies predominantly influence the order (or task) holons, the presented *Agenda Manager* component for resource holons will function similarly for both strategies.

The *Agenda Manager* component is responsible for managing the service provided by the resource holon. The component manages a list of service bookings by order holons and triggers the *Execution* component, with the necessary execution information, according to the agenda.

The *Agenda Manager* component requires two functions – one to receive and evaluate messages from the other holon components, and one to manage the resource's service bookings and execution. For handling the messages, a process running a *receive-evaluate* loop similar to that of the *Communication* component can be used. The messages are passed on to the process which manages the service.

The logic for the service management could be implemented in different ways. The logic can be implemented in a normal Erlang process or OTP behaviours can be used. OTP provides two useful behaviours – a generic server (*gen_server*) and a generic finite state machine (*gen_fsm*). The logic can thus be implemented in any of the mentioned ways, with the selection based on the approach which best matches the requirements of the service management model. A general summary of the *gen_fsm* behaviour library is provided in Appendix A.1.

4.2.2.3 Execution Component

The *Execution* component of the holon is responsible for driving the hardware actions related to the service of the resource holon. This component activates the execution of hardware functions, with the necessary execution information and in a specified sequence, to perform the service of the holon.

The *Execution* component is implemented similarly to the *Agenda Manager* component, i.e. a *receive-evaluate* loop process, for receiving messages, and a process for managing the service execution. The service execution can again be done in different ways, but using the finite state machine (FSM) behaviour is an attractive solution as the execution of resource holon services can usually be easily modelled as FSMs.

When using the FSM approach, the required sequence of execution actions is formulated into the *gen_fsm* behaviour. With each execution state, the necessary activation and information messages are sent to the hardware via the *Interfacing* component. The process receives feedback regarding the execution status from the hardware, which trigger the transitions between the states. When execution is completed, the execution result is replied to the *Agenda Manager* component, from where it is forwarded to the *Communication* component and ultimately replied to the order holon.

4.2.2.4 Interfacing Component

The *Interfacing* component maintains the communication interface between the Erlang control programs and the hardware. This component isolates the hardware specific communication structures from the execution logic.

The *Interfacing* component can be done in two ways, i.e. using OTP functions or using ports (or linked-in port drivers). When using the first approach, the component is implemented by a *receive-evaluate* loop process and a process implementing the OTP libraries for interfacing, such as *gen_tcp* or *gen_udp* (for TCP/IP or UDP communication). With the linked-in port driver approach, a program can be developed in another language (C, Java, etc.) and be wrapped with Erlang. The program can then be used as if it is just a pure Erlang module. This allows for the creation of communication structures which are not incorporated in OTP (such as Profibus or CANbus) or the use of a device specific driver or application programming interface (API). The use of ports and other Erlang/OTP integration tools is discussed in detail by Logan, Merrit, and Carlsson (2011).

Erlang also supports the use of eXtensible Markup Language (XML), which is frequently used with TCP/IP communication. Two popular libraries for XML functionality are XMErL (Anonymous s.a. [b]) and ErlSom (De Jong 2007). These libraries can be used, in conjunction with *gen_tcp*, to build and parse XML strings and files for use in socket communication.

4.2.3 Applicability to other PROSA holons

The presented methodology can be extended to the other PROSA holons. As all holons (and holon functional components) communicate through an exchange of messages, the communication process presented in section 4.2.1.3 can be applied. The process can be adapted for each specific holon component, according to the messages that may be received.

The *gen_server* and *gen_fsm* OTP behaviours are equally useful in representing the logic of the other holon types. These behaviours are especially applicable to the functionality of the order holon where service bookings must be managed along with task executions.

4.3 Case Study

As a case study, a resource holon for a pick-‘n-place robot was implemented using Erlang/OTP. This section describes the implementation of the functional components.

4.3.1 Communication Component

The *Communication* component is implemented as a single *receive-evaluate* loop process. Messages are received and forwarded according to a successful pattern match. To facilitate

the communication, a record was created for service-related messages. This record is constructed as follows:

```
#service{message_type, service_type, reply_to, conversation_ID, requester_pid, provider_pid, result, info}
```

- `message_type` - specification of service message, e.g. request, cancel, start.
- `service_type` - service specification, e.g. pick-'n-place, inspect, transport.
- `reply_to` – holon process ID to which reply must be sent (for inter-holon communication)
- `conversation_ID` - unique reference to the sequence of messages
- `requester_pid` – process ID of the requesting process linked to the service message
- `provider_pid` – process ID of the providing process linked to the service message
- `result` - Boolean result of action linked to service message
- `info` - information linked to the service message

The following code snippet shows the working of the *receive-evaluate* process of the *Communication* component (in this example named *robot_comm*), as pattern matching is used to distinguish between an intra-holon message (from the *Agenda Manager* component) and an inter-holon message (from another holon):

```
rec_messages() ->
	receive
		%message from agenda_manager in reply to service request
		{agenda_manager_fsm, Message=#service{}} ->
			%extract the corresponding process ID
			Pid = Message#service.reply_to,
			%send response to holon
			Pid ! {robot_comm, Message},
			%loop again
			rec_messages();

		%SERVICE message from other holon requesting a service
		{From, Message=#service{}} ->
			%forward message to agenda_manager
			agenda_manager_fsm ! {robot_comm, Message},
			%loop again
			rec_messages()
	end.
```

4.3.2 Agenda Manager Component

Two processes are used to implement the Agenda Manager component – one for handling communication and one for managing the holon service. The communication is handled by a process similar to that described for the *Communication* component. To manage the service, a process using the OTP behaviour for a generic finite state machine was chosen.

The state diagram used in the *Agenda Manager* FSM is shown in Figure 4. The states of the FSM each constitute two elements: execution status and a list of bookings (combined as a tuple in Figure 4). The execution status reflects whether the holon hardware is currently in operation ("busy") or idle ("free"), while the booking list keeps record of commitments made to requesting holons. The state transitions are driven by messages received from either the *Execution* or *Communication* components.

Code snippets from the *Agenda Manager* FSM are shown below. The code shows how events (which in these cases are the arrival of messages) are handled according to the specific state and how state transitions are specified. The presented code implements the states, events and

transitions highlighted in Figure 4. The handling of two different messages is shown when the *Agenda Manager* FSM is in the "free" state – the messages are of types "booking request" and "start", received from order holons. The code also shows the handling of a "done" message from the *Execution* component of the robot holon, in the "busy" state.

```
%STATE: free_booked --> resource is idle, but is booked
free_booked(Message=#service{message_type=booking_req},[Job_list]) ->
      %add request to bookings list
      NewJob_list=lists:append(Job_list, [Message#service.requester_pid]),
      %reply request result to Order holon via robot_comm
      robot_comm ! {agenda_manager_fsm,Message#service{result=true}},
      %specify the next state and state information
      {next_state, free_booked, [NewJob_list]};

%STATE: free_booked --> resource is idle, but is booked
free_booked(Message=#service{message_type=start},[Job_list]) ->
      %forward "start" message to resource_exec
      robot_exec ! {agenda_manager_fsm,Message},
      %specify the next state and state information
      {next_state,busy_booked,[Message#service.requester_pid,
      lists:delete(Message#service.requester_pid, Job_list)]}.


%STATE: busy_booked --> resource is busy and is booked
busy_booked(Message=#service{message_type=done},[CurrJob,Job_list]) ->
      %forward result message to Order holon via robot_comm
      robot_comm ! {agenda_manager_fsm,Message},
      %specify the next state and state information
      {next_state,free_booked,[Job_list]}.
```

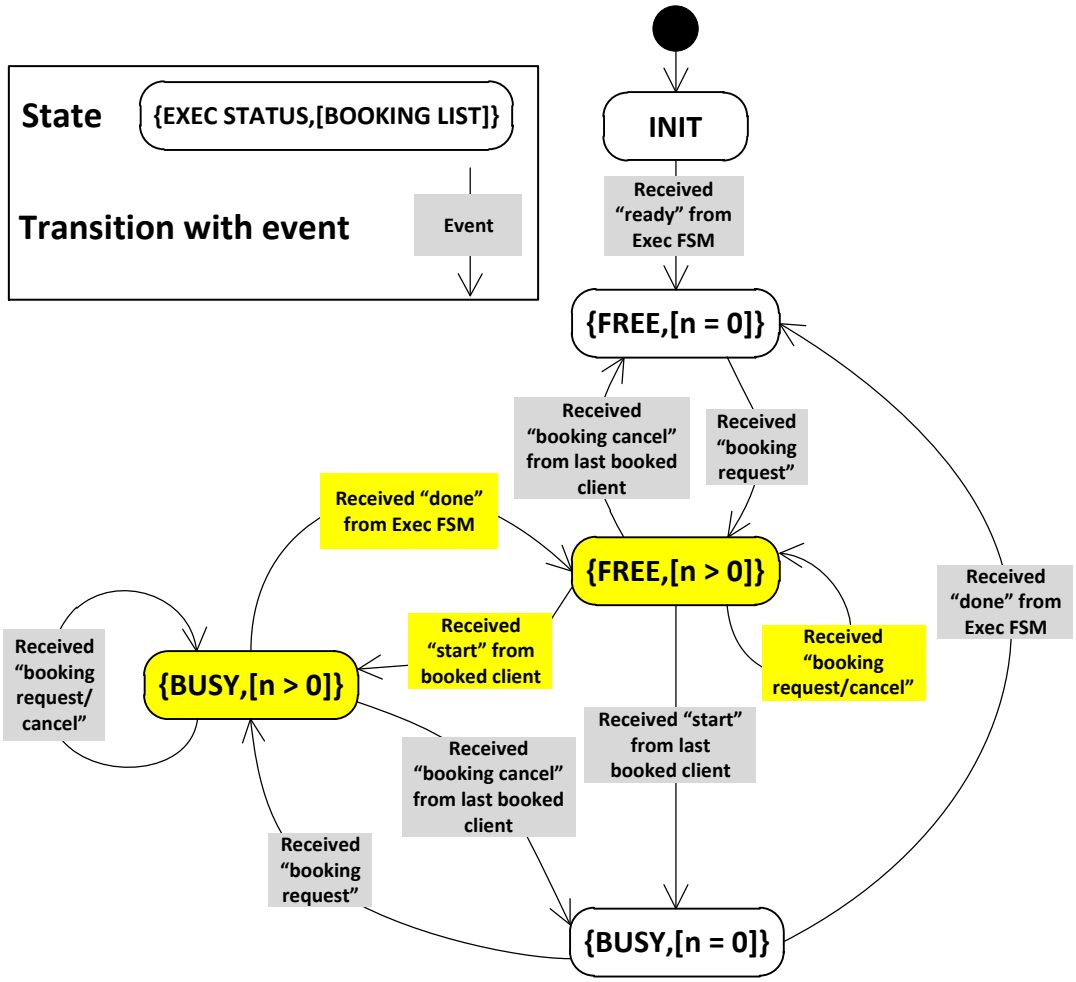

Figure 4. State diagram of the *Agenda Manager* FSM.

4.3.3 Execution Component

The *Execution* component is implemented similar to the *Agenda Manager* component – one process for handling communication and a *gen_fsm* process for managing the execution.

Figure 5 shows a simple example of an execution state diagram for the pick-'n-place robot holon. This example shows three states: "ready", "picking" and "placing" – each representing an execution state of the robot. The FSM switches between states in accordance with received messages from the *Agenda Manager* and the hardware.

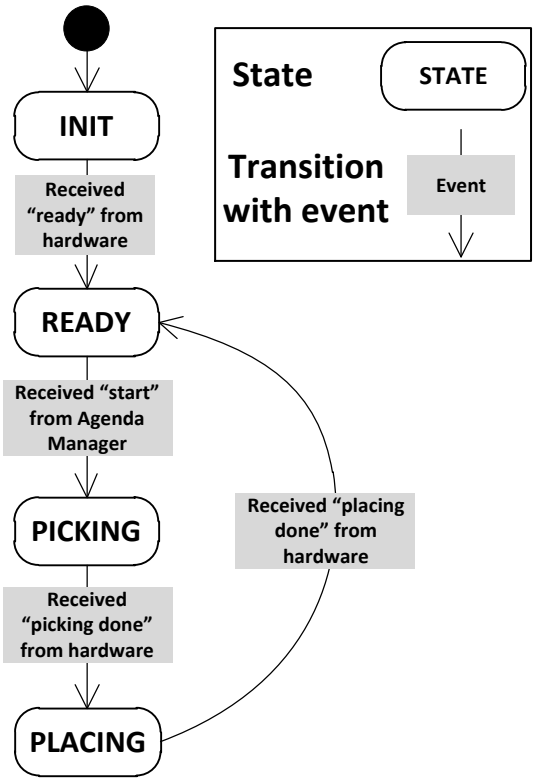

Figure 5. Example state diagram of the *Execution* FSM.

The implementation of the state diagram of Figure 5 using the *gen_fsm* OTP behaviour is shown by the following code snippet:

```
%STATE: ready --> ready to perform operation
ready(Message=#service{message_type=start},_) ->
      %send picking coordinates to interfacing component
      robot_pi ! {robot_exec, Message#service.info.coords.pick_coords},
      %specify the next state and state information
      {next_state, picking, Message}.

%STATE: picking --> executing picking operation
picking(picking_done, Message) ->
      %send placing coordinates to interfacing component
      robot_pi ! {robot_exec, Message#service.info.coords.place_coords },
      %specify the next state and state information
      {next_state, placing, {CurrJob, Message}}.

%STATE: placing --> executing placing operation
placing(placing_done, Message) ->
      %send result to agenda manager component
      agenda_manager ! {robot_exec, Message=#service{result=true}},
      %specify the next state and state information
      {next_state, ready, []}.
```

4.3.4 Interfacing Component

For the case study implementation, the control software of the resource holon interfaced with the controller of the robot through TCP/IP communication. The XMErL library is used to build and parse XML strings. The following code snippet shows how the *gen_tcp* OTP library (briefly summarized in Appendix A.2) is used to communicate to the robot controller:

```
socket_client(Info) ->
	%connect to TCP server
	{ok,Socket} = socket_connect(),
	%build XML string
	XML_string = build_XML(Info),
	%send string
	ok = gen_tcp:send(Socket, XML_string),
	%receive result of operation
	{ok,XML_data} = do_receive(Socket,[]),
	%close socket connection
	ok = gen_tcp:close(Socket),
	%extract result from string
	{XML_doc,_} = xmerl_scan:string(XML_data,[{encoding,latin1}]),
	Msg = extract_content('RESULT',[XML_doc]),
	Message=list_to_atom(Msg),
	Message.

socket_connect() ->
	%connect to socket
	case gen_tcp:connect(?address, ?port, [list,{packet,0},{active,false}]) of
		%success – return socket reference
		{ok, Socket} -> {ok, Socket};
		%failure – try again
		_ -> timer:sleep(1000),
		     socket_connect()
	end.
```

4.3.5 Typical operation scenario

To illustrate the sequence of functionality of the presented Erlang based robot holon, the operations involved in a typical scenario will be explained. The scenario entails the receiving of a *start* message from some order holon, i.e. a request from an order holon for the robot holon to start a previously booked service. This scenario was selected as it involves functions from all of the robot holon components.

For the explanation of the of the scenario it is necessary to describe the state of the holon FSM components. Assume that the Agenda Manager FSM is in the "free_booked" state – i.e. the robot holon is currently idle, but its service has been booked for use in the near future by order holons. The *Execution* FSM is in the initial "ready" state, awaiting a *start* message from the *Agenda Manager* to execute a pick-'n-place service.

When the physical part associated with the order holon is in the position for the pick-'n-place service (which was previously booked by the order holon) to be executed, the order holon will request the execution to be started by sending a *start* message to the *Communication* component of the robot holon. As is presented in section 4.3.1, the *Communication* component continuously awaits the arrival of a message through the *receive* function. When the order holon sends the *start* message, the message is received by the *Communication* components and is compared against the defined message patterns. The start message will match the following pattern:

```
%SERVICE message from other holon requesting a service
{From, Message=#service{}} ->
       %forward message to agenda_manager
       agenda_manager_fsm ! {robot_comm, Message},
       %loop again
       rec_messages()
```

Upon matching the pattern, the *Communication* component will forward the message to the *Agenda Manager* FSM component. The *Agenda Manager* FSM is in the "free_booked" state, thus the *start* message forwarded from the *Communication* component will be compared to the defined state transition patterns. The message will match the event specified by the following transition pattern:

```
%STATE: free_booked --> resource is idle, but is booked
free_booked(Message=#service{message_type=start},[Job_list]) ->
       %forward "start" message to resource_exec
       robot_exec ! {agenda_manager_fsm,Message},
       %specify the next state and state information
       {next_state,busy_booked,[Message#service.requester_pid,
       lists:delete(Message#service.requester_pid, Job_list)]}.
```

The *Agenda Manager* FSM will trigger execution of the service by forwarding the message to the *Execution* component, then transition to the next state "busy_booked". The internal state data of the FSM is also changed – the process ID of the order holon is removed from the list of received bookings and rather stored as an additional variable `CurrJob` (indicating the PID of the order holon involved in the current service execution) in the state data tuple.

The *Execution* component receives the *start* message as an event in the "ready" state (as shown in the code snippet of section 4.3.3) and proceeds to execute the pickup action of the pick-'n-place service by sending a message – containing the pickup coordinates as stored in the `info` field of the message from the order holon – to the *Interface* component. The *Execution* FSM then transitions to the "placing" state.

The *Interface* component extracts the coordinate information from the message received from the *Execution* component, builds an XML string and sends it to the physical robot controller using the *gen_tcp* library functions. As the robot completes the pickup action, an XML message is sent to the *Interface* component where the message is parsed and sent to the *Execution* component as the Erlang atom `picking_done`.

The interaction between the *Execution* and *Interfacing* components continue as described above until all the actions of the service have been completed – in this scenario, when the *Interfacing* component sends the atom `placing_done` to the *Execution* component. Before the *Execution* component then transitions back to the "ready" state (awaiting a *start* message for the next service execution), it sends a *done* message to the *Agenda Manager* FSM.

The *Agenda Manager* FSM will receive the *done* message from the *Execution* component in the "busy_booked" state. With the *done* message event, the *done* message is forwarded to the *Communication* component (which will use the associated PID field of the message to forward the message to the correct Order holon), before transitioning to the "free_booked" state.

4.4 Additional Erlang/OTP functionality

In addition to the OTP functionality used in the holon implementation described above, two further tools offered by Erlang/OTP can be very useful, i.e. the *Supervisor* and *Logging* modules.

Through the *Supervisor* module, Erlang allows the implementation of supervision trees, in the form of a process structuring model in terms of workers and supervisors. Worker processes do the computational work, while supervisor processes monitor worker processes. This hierarchical structure allows for the development of fault-tolerant programs, since supervisor processes can start and stop worker processes, and restart them if they should fail (Anonymous s.a. [a]).

As fault-tolerance is an important requirement for the modern manufacturing environment, supervision trees can be very advantageous. For the implementation of a resource holon, all the components discussed in the previous sections will be worker processes and can be supervised by a supervisor process. Upon starting, the supervisor process launches the processes in a specified order. The order to which they are terminated during shut down is also specified. A restart strategy can be specified for the supervisor process, i.e. the way in which processes are restarted in event of a process failing. Three options are available (Anonymous s.a. [a]):

- "one-for-one" – only the process that fails is restarted.
- "one-for-all" – if a worker process fails, all of the supervised processes are terminated and restarted.
- "rest-for-one" – if a worker process fails, it and the subsequent processes (in the start order) are terminated and restarted.

A supervisor process can thus be a very useful addition to the holon implementation. At the very least, it provides a neat and simple way to start and stop all the holon processes. With the selection of an appropriate restart strategy, a supervisor process can add great robustness to the holon implementation.

Logging modules offer useful functionality related to diagnosibility, an important requirement for reconfigurable systems. In terms of software diagnosibility, logging is an important tool. Erlang/OTP includes an *error_logger* module (Anonymous s.a. [c]) which can be used to output error, warning and information reports to the terminal or to file. The format of these reports can be customized according to the needs of the application. The *error_logger* module can be used by all holon processes to log events, errors and general process information to file, e.g. received and sent message information, state transitions and process failures. This information can be helpful for debugging or problem identification, or just for monitoring.

5 Conclusion

Reconfigurable manufacturing systems (RMSs) are intended for situations characterised by short product life cycles, large product variety and fluctuating product demand, since RMSs have the ability to reconfigure hardware and control resources to rapidly adjust the production capacity and functionality. RMSs commonly employ holonic control architectures, because they share many characteristics.

This paper motivates why the functional programming language Erlang and the Erlang-based OTP (Open Telecom Platform) present an attractive solution for implementing holonic

control. It is shown that the requirements for which Erlang was developed are highly relevant to holonic and reconfigurable control. The paper then presents an implementation methodology and case study using Erlang/OTP.

The presented case study for the Erlang/OTP implementation focusses on the resource holon, as defined by PROSA (Product-Resource-Order-Staff Architecture). A generic model for a resource holon to suit an Erlang implementation is presented, with four functional holon components, i.e. communication, agenda manager, execution and interfacing. The implementation of these components, using Erlang/OTP processes, is described.

Future work will entail the expansion of the Erlang/OTP implementation to the control system for an entire manufacturing cell, in which all of the PROSA holons will be incorporated. The Erlang/OTP manufacturing cell will then be subjected to a series of experiments – the results of which will be used to perform a quantitative and qualitative comparison with an equivalent MAS implementation.

6 References

Almeida, F.L., Terra, B.M., Dias, P.A., and Gonçales, G.M., 2010. Adoption Issues of Multi-Agent Systems in Manufacturing Industry. *Fifth International Multi-conference on Computing in the Global Information Technology*. pp. 238-244.

Anonymous, s.a. (a) *Get Started with OTP*. [Online]. Available: http://www.erlang.org (18 July 2013).

Anonymous, s.a. (b). *XMErL Reference manual*.. [Online]. Available: http://www.erlang.org/doc/apps/xmerl (28 March 2014).

Anonymous s.a. (c). *Erlang Kernel Reference Manual*. [S.a.]. [Online]. Available: http://www.erlang.org/doc/apps/kernel (28 March 2014).

Anonymous s.a. (d). *Erlang/OTP System Documentation.* [S.a.]. [Online]. Available: http://www.erlang.org/doc/pdf/otp-system-documentation.pdf (28 March 2014).

Armstrong, J., 2003. *Making Reliable Distributed Systems in the Presence of Software Errors*. Doctor's Dissertation. Royal Institute of Technology, Stockholm, Sweden.

Armstrong, J., 2007. *Programming Erlang: Software for a Concurrent World*. Raleigh, North Carolina: The Pragmatic Bookshelf.

Bi, Z.M., Wang, L., and Lang, S.Y.T., 2007. Current Status of Reconfigurable Assembly Systems. *International Journal of Manufacturing Research*, Inderscience. Vol. 2, No. 3: 303 - 328.

Bi, Z.M., Lang, S.Y.T., Shen, W., and Wang, L., 2008. Reconfigurable Manufacturing Systems: The State of the Art. *International Journal of Production Research*. Vol. 46, No. 4: 967 - 992.

Chirn, J.L. and McFarlane, D., 2000. A Holonic Component-based Approach to Reconfigurable Manufacturing Control Architecture. *Proceedings of the International Workshop on Industrial Applications of Holonic and Multi-Agent Systems*. pp. 219–223.

Däcker, B., 2000. *Concurrent Functional Programming for Telecommunications: A Case Study of Technology Introduction*. Master's Thesis. Royal Institute of Technology, Stockholm, Sweden.

De Jong, W., 2007. *Erlsom*. [Online]. Available: http://erlsom.sourceforge.net (28 March 2014).

ElMaraghy, H., 2006. Flexible and Reconfigurable Manufacturing System Paradigms. *International Journal of Flexible Manufacturing System*. Vol. 17: 61-276.

Graefe, R. and Basson, A.H., 2013. Control of Reconfigurable Manufacturing Systems using Object-Oriented Programming, *Proceedings of the 5th International Conference on Changeable, Agile, Reconfigurable and Virtual Production (CARV2013)*. pp. 231-236.

Hebert, F., 2014. *Learn Some Erlang For Great Good*. No Starch Press.

Holvoet, T. and Valckenaers, P., 2006. Exploiting the Environment for Coordinating Agent Intentions. *AAMAS Conference*. Hakodate, Japan (8–12 May).

Leitao, P. and Restivo, F.J., 2006. ADACOR: A Holonic Architecture for Agile and Adaptive Manufacturing Control. *Computers in Industry*. Vol. 57, No. 2: 121-130.

Kotak, D., Wu, S., Fleetwood, M., and Tamoto, H., 2003. Agent-Based Holonic Design and Operations Environment for Distributed Manufacturing. *Computers in Industry*. Vol. 52: 95–108.

Leitao, P. and Restivo, F.J., 2002. A Holonic Control Approach for Distributed Manufacturing. *Knowledge and Technology Integration in Production and Services: Balancing Knowledge and Technology in Product and Service Life Cycle*. pp. 263–270. Kluwer Academic Publishers.

Logan, M., Merrit, E., and Carlsson, R., 2011. *Erlang and OTP in Action*. Stamford: Manning Publications Co.

Mehrabi, M.G., Ulsoy, A.G., Koren, Y., 2000. Reconfigurable Manufacturing Systems: Key to Future Manufacturing. *Journal of Intelligent Manufacturing*. Vol. 13: 135-146.

Paolucci, M. and Sacile, R., 2005. *Agent-Based Manufacturing and Control Systems*. London: CRC Press.

Scholz-Reiter, B. and Freitag, M., 2007. Autonomous Processes in Assembly Systems. *Annals of the CIRP*. Vol. 56: 712–730.

Van Brussel, H., Wyns, J., Valckenaers, P., Bongaerts, L., and Peeters, P., 1998. Reference Architecture for Holonic Manufacturing Systems: PROSA. *Computers in Industry*. Vol. 37: 255–274.

Vinoski, S., 2007. Concurrency with Erlang. *IEEE Internet Computing*. Vol. 11, No. 5: 90-93.

Vyatkin, V., 2007. IEC 61499 Function Blocks for Embedded and Distributed Control Systems Design. *North Carolina: Instrumentation, Systems and Automation Society, ISA.*

Appendix A: OTP Libraries

This appendix provides a summary of the functionality and programmatic implementation of the OTP libraries mentioned in this paper. The presented work made use of two OTP libraries, namely the generic finite state machine (gen_fsm) behaviour and generic Transmission Control Protocol (gen_tcp) libraries. The description of the gen_fsm library is adapted from Anonymous s.a. [d] and, for the gen_tcp library, from Anonymous s.a. [c] and Hebert (2014).

A.1 Generic finite state machine behaviour library

A finite state machine can be described as a set of relations between states, events and actions. These relations can be expressed in the following form:

```
State x Event → Action(s), NextState
```

This expression states that when the FSM is in some `State` and some `Event` occurs, some `Action(s)` will be performed and the FSM will transition to `NextState`. Using the Erlang gen_fsm behaviour, these state transitions can be implemented by:

```
StateName(Event, StateData) ->
     %code for actions here
     {next_state, NextStateName, NewStateData}.
```

The name of the state the FSM is in when Event occurs is programmed as `StateName`. `StateData` represents internal information regarding the current state. When `Event` occurs, specific actions that must be performed can be programmed. After all the required actions are completed, the statement ends with a description of the state transition that follows. The transition description is represented as a tuple with three elements: the first element is the atom `next_state`, designating the transition description; the second element specifies the name of the state to which the FSM will transition to and the last element specifies the internal information associated with the next state.

The following code starts a gen_fsm behaviour in a new process:

```
gen_fsm:start_link({local, FsmName}, ModuleName, InitData, Options)
```

The variables have the following meaning:

- `FsmName` – the name by which the FSM process will be registered.
- `ModuleName` – the name of the module where the callback functions of the FSM (i.e. the functions defining the state transitions) are located.
- `InitData` – information passed to the FSM during initialization.
- `Options` – a list of possible options for the gen_fsm process – e.g. timeouts, debugging functions, etc.

When the gen_fsm behaviour is started, it enters the initialization function of the FSM, programmed as:

```
init(InitData) ->
     %code for initialization actions here
     {ok, InitialStateName, StateData}.
```

The function performs the necessary initialization functions and concludes with the definition of the initial state of the FSM. The FSM will consequently transition to `InitialStateName` with the accompanying `StateData`.

With the FSM now occupying a specific state, it can receive notifications regarding the occurrence of events. Processes can notify a specific gen_fsm process of an event using the following function:

```
gen_fsm:send_event(FsmName, Event)
```

This function constructs a message of the `Event` and sends it to the gen_fsm process. The event is handled in the current state of the FSM and will result in some corresponding state transition, as was discussed earlier in this section.

A.2 Generic Transmission Control Protocol library

The gen_tcp library included in OTP provides functions to communicate with sockets using Transmission Control protocol (TCP). Functions are included for both server and client implementations – the simplest forms of which are briefly presented in this section.

An Erlang process can act as a server for a designated TCP port, using:

```
{ok, Socket} = gen_tcp:listen(Port, Options)
```

- `Port` – the port number for the socket.
- `Options` – a list of socket configuration options.
- `Socket` – data type representing the TCP socket.

As the function name suggests, the server process will listen for incoming connection requests at the specified port. When such a request is received, the connection can be accepted:

```
gen_tcp:accept(Socket)
```

Also, a process can connect to a TCP socket as a client – this functionality is provided through the function:

```
gen_tcp:connect(Address, Port, Options)
```

- `Address` – the IP address or host name for the socket.

When the connection is accepted by the corresponding server process, TCP communication over the connected socket can be achieved. Both the server and client processes use the same functions for the sending and receiving of messages over the socket:

```
gen_tcp:send(Socket, DataPacket)
```

- `DataPacket` – information to be sent over socket.

```
gen_tcp:recv(Socket, Length)
```

- `Length` – the number of bytes to read from the socket.