Structured communication for concurrent languages

Sven-Olof Nyström
Box 311, S-751 05 Uppsala, Sweden
Phone: +46–18–18 25 00
Fax: +46–18–51 19 25

Abstract

I propose higher-level primitives for concurrent programming in Erlang. The first construct is for the rather common situation where a message is a request for information, and the receiver of the message is supposed to reply to the sender. The second construct is intended to allow a form of process definition which appears to be very common in Erlang programs to be expressed in a simplified form. These primitives are intended to allow typical Erlang program to be expressed in a more clear and compact manner, and can be introduced without modifications to existing run-time systems. The use of these primitives could also be helpful in the development of a type system for Erlang processes, and makes it feasible to develop a static analysis of Erlang programs.

1 Introduction

Message-passing is the goto of concurrent programming. It can be implemented efficiently and has an intuitive semantics, when seen in a small context. However, like goto, message-passing does not give the programmer much aid when organising larger systems.

Modern conventional programming languages offer higher-level control constructs (for example conditionals, loops and procedures) which are implemented at the machine level using goto. The use of these constructs makes the organisation of a program clearer, and is helpful in program development and maintenance. At the same time it is important to remember that the higher-level control constructs have a straightforward mapping to machine instructions, so the usefulness of these constructs does not lie in increased power, but in improved clarity.

Is it possible to, in a similar manner, find higher-level constructs for structured communication in concurrent languages? In this paper, we propose
two constructs for structured communication in asynchronous concurrent languages (such as Erlang [1]). Neither construct requires any modifications to the runtime systems of existing implementations, and programs using the constructs would still be able to communicate with traditional Erlang programs.

The first construct is for the rather common situation where a message is a request for information, and the receiver of the message is supposed to reply to the sender. Many process definitions have a very simple structure, and the second construct is intended to capture a form of process definition which appears to be very common in typical Erlang programs.

2 Requests and answers

Many messages are requests for information, e.g., for data stored by another process. Consider the following example, which loosely follows [1, chapter 5].

**Example 2.1** A counter process maintains the state of a counter. It can react to the following messages.

\[
\text{increment \hspace{1em} value \hspace{1em} stop}
\]

The increment message causes the counter process to increment its internal counter by one, the value message is a request for the current value of the counter, and the stop message causes the process to halt.

To implement the response of the value message in Erlang a simple programming idiom is used. The exchange of messages is described below, where the process $A$ is the one asking for information and the process $B$ is the responding process, e.g., the counter process in the example.

1. The process $A$ sends the request to process $B$. In the request the process identifier of $A$ is included.

2. The receiver (process $B$) recognises the message, extracts the process identifier and computes the answer.

3. Process $B$ sends the answer to to $A$. To make sure that $A$ can recognise the answer, the process identifier of $B$ is included in the answer.

4. Process $A$ recognises the answer by comparing the included process identifier with $B$'s process identifier.

To program this sequence of interaction in Erlang is quite straight-forward, but it is still natural to wonder if there is an easier way to let process $A$ ask process $B$ a question.
2.1 A review of Erlang notation

Before we turn to the proposed communication primitives a brief review of
the current communication primitives of Erlang.

Messages are sent using the ‘!’ primitive:

\[ \text{Pid } ! \text{ Message} \]

The first argument of ‘!’ is an expression which evaluates to a process iden-
tifier, and the second part an expression which evaluates to the message.

The \texttt{receive} primitive of Erlang reads a message and matches it against
a sequence of patterns. For example,

\begin{verbatim}
receive
    increment ->
        ...
    value ->
        ...
    stop ->
        ...
end
\end{verbatim}

reads a message from the input, matches it against the three alternatives,
and branches to the code associated with the first match.

2.2 A proposal

We propose the following notation.

To send a request \texttt{Message} to process \texttt{Pid}, and then receive an answer in
the variable \texttt{Answer}, the syntactic form consists of a request followed by a
sequence of actions followed by a receive answer, e.g.,

\begin{verbatim}
request Pid ! Message,
    ...
receive_answer Answer,
    ...
\end{verbatim}

Here, the process sends the message \texttt{Message} to the process \texttt{Pid}, then ex-
ecutes some actions, and then waits for the answer to its request. After
\texttt{receive_answer} has been evaluated, the variable \texttt{answer} will be bound to
the answer.

The syntax allows a sequence of actions to occur between the request and
the \texttt{receive_answer}. This gives the programmer an opportunity to handle
problems with latency by letting the process continue its execution after it
has sent a request.

To reply to a request:
receive
    value ->
    ...
    answer Answer
    ...
end

Here, Answer is an expression which evaluates to the message which is sent
to original sender.

The answer primitive may only occur inside a receive and the corre-
sponding message must have been sent using the primitive request.

3 Advantages with the request-answer mechanism

The most obvious advantage is of course that a common communication
protocol can be expressed in a more compact and readable manner. Below
we discuss other advantages.

3.1 Implementation and efficiency

The simplest implementation of the request-answer constructs is to use the
existing message-passing mechanism, and use some tagging scheme to dis-
tinguish requests and answers from other types of messages. A more efficient
solution would be to use some special data structure to store the answer of
a request. This would be more efficient than the current situation, where
the sender of a request has to search the sequence of incoming messages (the
‘mailbox’) for the answer to the request.

3.2 Static analysis

Since it is no longer necessary for a process to send its identifier in order to get
an answer it is likely that the introduction of the request-answer mechanism
would make Erlang programs more amenable to program analysis.

3.3 Typing

Consider the sequence of incoming messages to a process $A$. Currently,
if $A$ asks a counter process about the state of the counter, the response
will show up in $A$'s mailbox among other messages to $A$, and there is no
general way to tell the different types of messages apart. By the request-
answer constructs, it is possible to distinguish between messages related to
the service $A$ provides to other processes, and the service provided to $A$ by
other processes.

The type of $A$ can now be given as the set of messages that are related to
the services $A$ provide. For example, the type of the counter process above
can be given as

\[
\text{increment value:integer stop}
\]
The value message is the only one that will give a response, which will always be an integer.

A more interesting example is a database system [1, chapter 9].

The process managing the database will react to three kinds of messages (for readability we deviate from Erlang syntax), \texttt{read(key)} will return the value stored at key \texttt{key}, \texttt{write(key,value)} will store the value \texttt{value} at key \texttt{key}, and \texttt{delete(key)} will remove the entry at key \texttt{key}. Suppose we have types \texttt{keyType} and \texttt{valueType} giving the types of the keys and values to be stored in the database. The type of the database process is

\begin{verbatim}
read(keyType): valueType
write(keyType,valueType)
delete(keyType)
\end{verbatim}

The type of the database process gives the external interface of the database. A database process may be distributed, and would then need to communicate with other processes in order to perform (say) a data base lookup. Such internal communication actions would not be a part of the external interface would thus not affect the type of the process.

In general, the type of a process gives its external interface. Given the type of a process, a programmer can at a glance tell which messages it expects. It is possible to discover erroneous messages at compile time.

4 Passing requests to other processes

The request-answer mechanism handles the situation where one process \texttt{A} sends a request to a process \texttt{B}, and the process \texttt{B} answers the request, but it cannot handle a situation where process \texttt{B} wants to pass the request to a third process. For example, in a distributed database it is natural to implement the database using a number of communicating processes, which pass database requests between each other.

To handle this situation we propose a primitive with the syntax

\begin{verbatim}
relay Pid ! Message
\end{verbatim}

with the simple rule that it may occur at exactly those positions in which an \texttt{answer} \ldots \ may occur. A \texttt{relay} will pass a request to the process \texttt{Pid}. The request received by \texttt{Pid} will be as indicated by \texttt{Message}.

For example, in the two-level distributed database described by Armstrong, Virding, and Williams [1, Section 9.3], the first-level server processes are responsible for distributing messages to the second-level servers. For the first-level servers to know where the data is stored, the keys are divided in two components, where the first component determines which second-level server is responsible for the storage of the corresponding data. When a first-level server receives a read message, it uses a local dictionary to determine
which second-level database to pass the message to. Using the relay primitive described above, the code for the first-level server might be as indicated by the following fragment.

```erlang
server(Dict) -->
    receive
        {read, {Key1, Key2}} -->
            case lookup(Key1, Dict) of
                {value, S2} -->
                    relay S2 ! {read, Key2};
                undefined -->
                    answer undefined
            end,
        server(Dict);

    ...;

end.
```

5 Servers

The second type of language construct we will consider is a form of process definition. Processes defined using this construct will be called servers. This might cause some confusion since ordinary Erlang processes are sometimes called servers, but we will stick to the name for now, since it is likely to lead the reader on the right path.

Many process definitions in Erlang have a very simple structure. They consist of a loop which reads a message, matches it against a number of possible patterns, and performs some action corresponding to the first matching pattern. We propose a new form of process definition to handle this special case more explicitly.

Let a definition of a server be as indicated by the following.

```erlang
server foo(X, ...)
    message1 ->
        ...
        foo(...);

    message2 ->
        ...
        foo(...);

    ...

    messageN ->
```
A server is a function which reads a message from input, finds the first matching pattern among the alternatives, and executes the corresponding actions. The sequence of actions must always end with a tail recursive call, giving the new state of the process, or an explicit stop, which causes the server to halt. A server must be started by a spawn, so the only other type of calls allowed to a server are the tail recursive ones.

The reason for requiring a stop is to allow easy detection of the situation where the programmer forgot a tail recursive call.

**Example 5.1** The counter process as a server.

```
server counter(Val)
    increment ->
        counter(Val+1);

    value ->
        answer Val,
        counter(Val);

    stop ->
        stop
end
```

A process that wants to know the value stored by the counter process C might do

```
request C!value,
receive_answer X,
...
```

which would bind the variable X to the current value of the counter. If the process has something useful to do while it is waiting for the answer, the following code is perhaps preferable.

```
request C!value,
    something_useful,
    more_useful_stuff,
receive_answer X
```

Many Erlang processes can be rewritten as server processes. A server process has a simpler dynamic behaviour and is thus easier to understand. Due to the lack of a control state, it should be possible to find more efficient implementations of server processes.
6 Related Work

A mechanism to allow a process to respond to a message was described by Brinch Hansen [2].

Kahn, Tribble, Miller and Bobrow [3] present a concurrent object-oriented programming language in which the processes read and respond to messages in a manner quite similar to the behaviour of server processes described above.

7 Conclusions

The proposed primitives can be introduced into existing Erlang implementations with no or small modifications of the run-time systems. They are not intended to be more powerful than the existing constructs, but to allow common programming idioms to be expressed more clearly.

Acknowledgements

Discussions with colleagues at the department of computing science have been helpful. Thomas Lindgren and Johan Bevemyr pointed out that it is common for Erlang processes to pass requests to other processes. This led to the relay primitive.

References

