# Using asynchronous tuple space access primitives (BONITA primitives) for process co-ordination

Antony Rowstron

Computer Laboratory, University of Cambridge, New Museums Site,
Pembroke Street, Cambridge CB2 3QG, UK

**Abstract.** In this paper an interactive talk program is used to demonstrate the difference between the Linda primitives and the recently proposed BONITA primitives. Both use the concept of shared tuple spaces for inter-agent communication, but the BONITA primitives provide asynchronous tuple space access. The paper demonstrates the performance gains and the novel co-ordination patterns achievable using the BONITA primitives.

## 1 Introduction

The concept of shared tuple spaces is the foundation of the Linda[1] co-ordination language. Linda provides asynchronous process communication but synchronous tuple space access. The BONITA[3] primitives are a set of asynchronous tuple space access primitives. In distributed environments the need for such primitives is driven by both functionality and performance concerns. The implementation of the C-BONITA uses the run-time system used in the York Linda Kernel II[2] and the primitives are described in detail in Rowstron et al.[3]. A detailed description of Linda can be found in Carriero et al.[1].

In order to compare the use of the BONITA primitives and the Linda primitives the implementation of a talk tool is considered. The talk program requires an arbitrary number of people should be able to communicate concurrently (interactively) using the talk program and the text that makes the conversation should be stored for future reference. The people involved in the conversation can dynamically alter. An initialisation section displays the conversation to date, and the main section allows the user to participate in the conversation.

### 1.1 C-Linda version

This initialisation code (lines 1–9, Figure 1) assumes that each conversation uses a unique tuple space (in this case represented by a tuple space handle called `con`), and this has been initialised and a tuple representing a counter of the number of messages in the conversation is present. Each line of the conversation is a simple tuple of the form $[index_{integer}, name_{string}, textline_{string}]$, and the counter tuple contains the value of `index` for the next line to be inserted.

```
1     out(con, user_name);
2     in(con, ?num_lines); /* Get counter */
3     out(con, num_lines, user_name, "Joining");
4     out(con, ++num_lines);
5     init_window();    /* Set up the window  */
6     for (pos = 0; pos < num_lines; pos++) {
7       rd(con, pos, ?name, ?text);
8       print_screen(name,text);
9     }
10    next = num_lines;
11    while (!exit_status) {
12      /* If available display next line */
13      if (inp(con, next, ?name, ?text)) {
14        print_screen(name,text);
15        next++;
16      }
17      if (ready_line(text)) {
18        in(con, ?num_lines);
19        out(con, num_lines+1);
20        out(con, num_lines, user_name, text);
21      }
22    }
23    in(con, user_name);
```

**Fig. 1.** The talk program using C-Linda.

The first operation is to insert a tuple containing the users name (line 1). This is so other users can ask who is currently active in a conversation. When a user exits the tuple containing the name is removed (line 16). A line in the conversation is inserted to indicate that the a new person has joined the conversation (lines 2–4). This is achieved by removing the counter tuple (line 2), and then inserting a new conversation line (line 3), and then replacing the incremented counter tuple (line 4). Then, each line of the conversation is read and printed on the screen (lines 6–9). The main section code (lines 10–23, Figure 1) of the C-Linda program uses polling for detecting the tuple containing the next line of the conversation in the tuple space and for getting user input text. An `inp` primitive is used to keep checking if a tuple containing a new line of the conversation has been inserted into the tuple space (line 13). If so, it is displayed and the local counter incremented thus enabling checking for the next tuple. The function `ready_line` manages the input of text from the keyboard and checks if a line of text is ready. If a line of text is ready it is inserted into the conversation by retrieving the counter tuple (line 18), reinserting it incremented (line 19), and then adding the tuple to the conversation (line 20).

### 1.2 C-BONITA version

The initialisation code (lines 1–12, Figure 2) for the C-BONITA version functionally does exactly the same as the C-Linda version. It should be noted that the BONITA primitives can emulate the Linda primitives, hence the Linda primitives are provided as macros. The second part of the initialisation code uses pipelining of tuple space accesses. The requesting of all tuples is achieved within the for loop (line 7) and the `dispatch` primitive (line 8). The *request identifier* for each of the `dispatch` primitives is stored, to be used when retrieving the results. The for loop (line 9) and the `obtain` primitive (line 10) retrieve the requested tuples. The C-BONITA main section code (lines 13–29, Figure 2) again uses polling to check if the next line of the conversation is available or if there is user input ready. The polling of the next line of the conversation *does not* use `inp` but instead requests the tuple using a `dispatch` primitive (lines 14 and 20) and checks for the tuples arrival using the `arrived` primitive (line 17).

```
1     dispatch(con, user_name); /* Place name in ts */
2     in(con, ?num_lines); /* Get the line counter */
3     dispatch(con, num_lines, user_name, "Joining");
4     dispatch(con, ++num_lines);
5     init_window(); /* Set up the window */
6     /* Get lines of text - pipelining the ts access */
7     for (pos = 0; pos < num_lines; pos++)
8       arr[pos] = dispatch(con, pos, ?n, ?text, NONDEST);
9     for (pos = 0; pos < num_lines; pos++) {
10      obtain(arr[pos]);
11      print_screen(n,text);
12    }
13    next = num_lines;
14    ref = dispatch(con, next, ?n, ?text, NONDEST);
15    while (!exit_status) {
16      /* If next line here then display it  */
17      if (arrived(ref)) {
18        print_screen(n,text);
19        /* Request the next line of the conversation */
20        ref = dispatch(con, ++next, ?n, ?text, NONDEST);
21      }
22      if (ready_line(text_input)) {
23        in(con, ?num_lines);
24        dispatch(con, num_lines+1);
25        dispatch(con, num_lines, user_name, text_input);
26      }
27    }
28    in(con, user_name);
```

**Fig. 2.** The talk program using C-BONITA.

It should be noted that the `dispatch` primitive on line 14 is used to request the first tuple containing a text line that was not displayed during the initialisation section, and subsequent tuples containing text lines are requested by the one on line 20. If a conversation tuple is available then the `arrived` primitive retrieves it (line 17), and the text displayed (line 18). The function `ready_line` is used to manage and check the text input by the user. If a line of text is available, the counter is retrieved (line 23), and then incremented and reinserted (line 24) and then the tuple with the line of text is inserted (line 25). At the end the users' name is removed from the tuple space (line 28).

## 2 Comparison of the C-Linda and C-BONITA programs

By examining the two programs it is clear to see that the C-Linda version appears more compact and in some ways more elegant. The fundamental difference between the two initialisation sections is the way in which the tuples representing the text of the past conversation is retrieved. The C-BONITA version pipelines access to the tuple space, whereas the C-Linda version does not. This cannot lead to the C-BONITA version taking longer, and in most cases will provide a speed-up. This is because when using a `rd` primitive, a message is sent to the system managing the tuple spaces (kernel), the kernel processes the message, and a reply message is sent back. When this is received the `rd` has completed and the program continues to perform the next `rd` primitive. In the C-BONITA version the the request for the next tuple is *not* dependent on the receipt of the result for the previous request.

The results shown in Table 1 show the time taken to retrieve a number of tuples from a tuple space. These were produced using a network of Silicon Graphics Indy workstations connected by a non-dedicated 10Mbit/s Ethernet LAN

network. The kernel used was the York Kernel II[2] but using it as a *single* centralised server, instead of using it as a distributed server (this favours the C-Linda version). The test programs were executed on a different Indy Workstation to the one which the kernel was executing on. The C-Linda test program was lines 6–9 of Figure 1, and the C-BONITA test program was lines 7–12 of Figure 2. The results for the C-BONITA version demonstrate the speed advantage of pipelining the tuple space accesses.

| Number of tuples | C-Linda (seconds) | C-BONITA (seconds) | C-BONITA speedup |
|---|---|---|---|
| 1 | 0.004 | 0.004 | 0 (0%) |
| 10 | 0.033 | 0.019 | 0.014 (42%) |
| 100 | 0.362 | 0.197 | 0.165 (46%) |
| 1000 | 4.645 | 2.683 | 1.962 (42%) |

**Table 1.** Timings for retrieving tuples.

The fundamental difference between the two main sections is the way in which the tuples representing the next line of text in a conversation are retrieved. In the C-Linda version this is achieved using the `inp` primitive and in the C-BONITA version this is achieved using a `dispatch` and `arrived` primitives. The C-Linda approach has a disadvantage over the C-BONITA approach due to the level of communication between the user process and the kernel. The C-BONITA approach requires two messages to pass between the user process and the kernel (dispatching the template and reply tuple). `Arrived` is simply a local check within the user process to see if the reply message has arrived, there is no extra communication with the kernel. An `inp` is two messages (a dispatch and a reply), but the reply will either contain a tuple or a value indicating no matching tuple was found. *Each time* an `inp` both messages are required. This also increase the load on the kernel, because it must process each message and create a reply.

## 3 Conclusions

We have briefly compared the Linda primitives and the BONITA primitives using a simple example, and shown that the BONITA primitives are better suited to the type of co-ordination required in agent systems. A full copy of the paper is available as Technical Report No. 422 from the Computer Laboratory, Cambridge University. The author would like to thank Dr. Andy Hopper and ORL Ltd, Cambridge for funding this work.

## References

1. N. Carriero and D. Gelernter. Linda in context. *Communications of the ACM*, 32(4):444–458, 1989.
2. A. Rowstron and A. Wood. An efficient distributed tuple space implementation for networks of workstations. In *Euro-Par'96*, LNCS 1123, pages 510–513. Springer-Verlag, 1996.
3. A. Rowstron and A. Wood. BONITA: A set of tuple space primitives for distributed coordination. In *HICSS-30*, volume 1, pages 379–388. IEEE CS Press, 1997.