Research report 180

MAKING CASSANDRA PARALLEL AND DISTRIBUTED

IAIN D. CRAIG

(RR180)

This brief set of notes outline ways in which the author's cassandra architecture can be implemented as a parallel or distributed system.

Department of Computer Science
University of Warwick
Coventry CV4 7AL
United Kingdom

March, 1991

MAKING CASSANDRA PARALLEL AND DISTRIBUTED

Iain D. Craig
Department of Computer Science
University of Warwick
Coventry CV4 7AL
UK EC
JANET: idc@uk.ac.warwick
TEL: 0203-52-3682

ABSTRACT

This brief set of notes outline ways in which the author's cassandra architecture can be implemented as a parallel or distributed system.

1. INTRODUCTION

These notes say very much what one might expect about how CASSANDRA can be turned into a distributed system. There are opportunities within the architecture for making it a parallel system, as well. Both of these are briefly outlined. We begin with how to make the architecture truly distributed. A formal specification (in Z) of the architecture appears in Craig (forthcoming): the specification shows how level managers are interfaced to a communictions medium. Some of the information contained herein has not appeared elsewhere. It is assumed that the reader is familiar enough with the literature to be able to locate, for example, papers on CAGE and POLIGON.

2. DISTRIBUTION

The basic method of distribution is extremely simple: allocate one processor to each level manager. This is not the only way of distributing the system: it would be better to state that each level manager be allocated to an independet process (where the processes actually reside then becomes a matter for confinguration management). Inter-level manager communications must be arranged so that the basic architectural constraints (those relating to uni-directional communications) are respected. We mention briefly how this can be done (note that we assume that the communications architecture being used is similar to that provided by SUN).

As part of each level manager, there is a communictions module which is responsible for managing all inter-level manager communications. In particular, it is charged with ensuring the correct connection of ports and with the management of messages. The basic

architecture does not mention aspects such as store-and-forward, so this is left as an option (the underlying comms system might do this transparently).

When a level manager is started, its communications module takes the port descriptions which define the level manager and attempts to connect the corresponding channel. It has also to verify that each port is only used in one direction. Next, it creates the channel and sends an initialisation message to the level manager at the other end of the channel (this operation is similar to the wake-up operation on Unix ports). This suggests that each channel is, in fact, bi-directional: after due consideration[1], this seems to be the best way of arranging things in CASSANDRA---for example, it allows 'out-of-band' messages to be sent in the opposite direction to perform such functions as, for example, warning the other level manager that a catastrophic failure has occurred. The out-of-band facility is invisible to the user and to the user level manager (it is entirely reserved for the communications module).

When communications have been established, level manager identifiers are exchanged: this is just a means of checking that the channel is correctly established. After the channel has been verified, some applications may need to exchange information of a relatively standard kind about their operations. This might include information about: physical location, tasks that can be handled, problems solvable, knowledge available, and so on. This information can be used to provide level managers with data on where and where not to send messages. This information may, ultimately, turn out to be highly problem-specific and is intended to be used by 'intelligent routers'.

After these exchanges, the level manager is able to attempt the solution of problems. In a distributed context, this amounts to the exchange of messages of two types (these are documented in Craig, 1989): data messages; control messages. Data messages contain the non-local information that is used in problem-solving. Control messages (which have not been explored to any real extent) deal with issues such as 'how are you getting on?', 'can you accept a problem of kind K?', etc., and the appropriate responses.

These messages, essentially, require the Knowledge Source interface described in Craig, 1989 and formalised in Craig (forthcoming). The interesting issues, here, relate to whether message buffering is used. In the formal specification, we assumed that the communications system implemented one method and that CASSANDRA merely adopted it; the use of message buffering will depend upon a variety of factors (bandwidth, nature of communications architecture, etc.). In short, there seems to be no particular solution which will apply to all systems, so we remain neutral on the issue. Within a level manager, it

[1]Which is not mentioned in Craig (forthcoming)---this is because it proved to be too low a level operation to be given a neat specification.

seems to be the case that buffering is not required on the output side; for incoming messages, the issue is rather more complex. Basically, when a message arrives, it must be routed either to the requesting Knowledge Source or to the scheduler: if the level manager is implemented as a serial program, messages can be lost (unless the communications architecture buffers them on behalf of the application and also unless it allows the application to request particular messages—this turns out to be important because CASSANDRA can prioritise messages). An internal message buffer seems to be appropriate (and is included in the formal specification).

As described in Craig, 1989, *all* communications between Knowedge Sources and the communications module are defined in terms of standard interfaces: the same applies to the local scheduler. In other words, the communications module should be totally encapsulated. This means that Knowledge Sources communicate with the communications module via pre-defined procedure calls.

The close-down procedure is the converse of the wake-up one.

3. PARALLEL IMPLEMENTATION

CASSANDRA was originally designed to be a parallel and distributed architecture. That is, it was to have been globally distributed with parallelism in the local components. The first design[2] contained the level manager construct. The level manager was, however, a collection of relatively independent parallel processes, each working concurrently on the solution of a problem. The parallel components (which were called Knowledge Sources) were to have been relatively simple and were to have been constructed out of simple elements. The relationship between each CASSANDRA Knowledge Source and the kind more familiar from blackboard systems was close, but not as close as in what became CASSANDRA. What follows is a reconstruction of that original design.

The central opportunity for parallelism within a level manager is seen to the the Knowledge Source. Each Knowledge Source contains three components: a trigger, a precondition and an action. Neither triggers nor preconditions can alter the state of the local memory, so they can be evaluated in parallel.

Triggers respond to purely local events (triggering on message arrival was banned from day one because it leads to too close a coupling between Knowledge Sources and communications—perhaps this close coupling might actually turn out to be desirable), and

[2]The design was abandoned because it turned out to be extremely difficult to implement on a VAX and in FranzLISP. There were, in addition, some theoretical problems which made this form of the architecture unattractive—it might be better with a new analysis.

they were permitted to examine the values stored in the entry which caused the event. The original view of triggers was that they resembled finite-state devices. Once an event occurred, the identifier of the entry would be broadcast to all Knowledge Sources. Those Knowledge Sources whose triggers were interested in an event of that kind would then examine some (maybe all) of the attributes of the entry. This evaluation would take place in parallel, and the result would be a set of Knowledge Source Activation Records (KSARs) of a traditional type: the KSARs would be inserted into a triggered list.

When a triggered list had been constructed, precondition evaluation could occur in the normal fashion. Precondition evaluation included sending messages to non-local destinations (just as in the final CASSANDRA architecture): this led to use of the Kleene Strong 3-valued interpretation for connectives, a property retained in the final architecture design. Since preconditions do not cause side-effects, they can be evaluated in parallel: in other words, *all* preconditions in the triggered list would be evaluated in parallel. To increase the parallelism in preconditions, their evaluation could be seen as a parallel walk of a binary tree (assuming that preconditions are represented in terms of conjunction, negation and disjunction). Failure propagation would occur in the usual way. When a precondition executed a message-sending primitive, its evaluation would be suspended until a reply had been received. There are variations on this theme, so it is not considered any further.

The result of precondition evaluation is a set of KSARs: these represent the Knowledge Sources that can be executed to cause local database changes. The execution of these KSARs was to have been controlled by the local scheduler. In a conventional blackboard system, the scheduler typically selects one KSAR to execute. In other words, the scheduler works as a kind of serialiser. In some contexts, it is desirable to execute more than one KSAR at a time, but this leads to problems in a concurrent environment (especially if deadlock is to be avoided). It seemed a pity to have all of that parallel activity end up in a simple serial program, so it was decided that Knowledge Source actions should be composed of production rules (in a fashion similar to CAGE). Because rule conditions do not cause side-effects, they, too, can be evaluated in parallel (using a mechanism similar to that used for preconditions). Another benefit of productions is that the actions they perform tend to be relatively fine-grained: thus, only relatively small regions of the local database would be affected by the execution of any one rule. Within a rule, execution would be sequential, but on a more global scale, rules could be executed in parallel.

The execution of Knowledge Source actions leads, of course, to the standard problem of concurrent updates. This turned out to be a significant problem for the architecture. On reflection, one solution to the concurrent updates might be to represent each attribute as an independent cell which belongs to a class of cells (this is similar to the POLIGON approach,

but is more radical). The updates problem for entries then reduces to the updates problem for cells, and it seems less likely that two Knowledge Sources will attempt to update the same cell at the same time (although this is an empirical problem—in any case, it seems unlikely that concurrent update of cells can be totally ruled out). The class mechanism is used as a device for limiting the operations that are available for each cell type.

The level manager could operate in a number of different modes. In the first mode, triggering and precondition evaluation represented phases that were sequentially composed. In other words, all triggers were evaluated in parallel (and this was performed concurrently with action execution); next all preconditions were evaluated in parallel. To make matters clearer, we use an OCCAM-like notation:

```
SEQ
    PAR[i=1 to numKSs]
        eval_trigger
    PAR[j=1 to numKSARs]
        eval_precond
    exec_ksar := select(ksars)
```

where *select* is the scheduler function and *ksars* is the list of KSARs produced by the precondition-evaluation process. *exec_ksar* is either one or a set of KSARs which are to be executed.

The second, and very much more radical, approach is to allow everything to run in parallel. This causes problems because it is not clear how to preform control under these circumstances.

This design represents the further definition of the contents of the level manager construct: it does not impact upon the global aspects of the architecture. Just as the formal specification of CASSANDRA revealed some interesting things that were new, a formal specification of an internally parallel CASSANDRA system might lead to definite opinions on some of the issues raised above.

REFERENCES

Craig, 1989. Craig, I.D., *The CASSANDRA Architecture*, Ellis Horwood, Chichester, 1989.

Craig, forthcoming. Craig, I.D., *The Formal Specification of Advanced AI Systems*, Ellis Horwood, Chichester, to appear, 1991.