Distributed Input Scheduling

0 Introduction

We consider a fixed and finite collection of concurrent processes. Each process produces one, potentially unbounded, *stream* of so-called *items*. This is called the *output* stream of the process. Also, each process consumes one, potentially unbounded, stream of items called the *input* stream of the process. Items are values of a fixed, but for our purposes irrelevant, datatype.

The items in the output stream are produced one-by-one, that is, the output stream is linearly ordered. Also, the items from the input stream are consumed one-by-one, that is, the input stream is linearly ordered too. Items in the output stream may depend on items from the input stream that have been consumed "earlier".

An item from the input stream may be delivered to a process before the process is ready to consume it; such items are not lost but are kept until the process is ready. The buffering required to implement this is considered given.

The problem to be solved is the construction of a distributed protocol for the computation of the input streams of the processes, for given processes and their output streams. These input streams must satisfy the following requirements:

0. The *same* input stream is delivered to *all* processes; that is, each process receives the same items in the same order as the other processes. As a result, we can speak of "the" (one and only) input stream of the processes.

1. Delivery of the items of the input stream to the processes is *not* required to be strictly *synchronous*; that is, one process may already have received more items in (its local instance of) the input stream than the others.

2. The input stream is the result of *interleaving* the output streams of the processes. This means:

   (a) Every item of the input stream occurs in one of the output streams.

   (b) Every item of every output stream occurs (exactly) once in the input stream.

   (c) Items from the same output stream occur in the same order in the input stream as in their output stream.

Requirements (a) and (b) state that the input stream contains the same items as the output streams, without duplication or omission. Requirement (c) states that interleaving is *order preserving*.

3. Obviously, an item can only be added to the input stream after it has been produced, that is, after one of the processes has added it to its output stream. As a requirement of *progress* such an item must occur in the input stream *soon enough*; more precisely: as soon as an item has been produced only a *limited* number of other items may be added to the input stream before this item is added to the input stream.

4. Every item in the input stream is delivered with a finite delay to all processes. This is a progress requirement too.

1 Communication channels

The processes do not have shared storage but are pair-wise connected by means of a communication medium. Via this medium every process can send a (so-called) *message* to every process, even to itself. A message may be an item from a stream but may also contain other data, depending on what is needed. A message may even contain no data at all, in which case it is called a *signal*.

The communication medium has the following given properties:

0. To send a message to a process, the sending process specifies (the contents of) the message and (the name of) the receiving process. A process receiving a message also receives the name of its sender.

1. Communication is *reliable*: every message sent will arrive without damage and no message will arrive before it has been sent.

2. Communication is *asynchronous*: a send action terminates immediately, that is, possibly before the message has arrived at its destination; as a consequence, arrival of a message carries no other information about the state of the sender than that it has sent that message.

3. A message may be sent before previous messages have arrived, so the connection between two processes allows simultaneous transmission of several messages. This connection does *not* have the First-In-First-Out property: it is not guaranteed that messages arrive in the same order in which they have been sent.

exercise

0. Design a protocol for message transmission from one process to another that does have the First-In-First-Out property, via a connection that does not have this property.

□

To evaluate the performance of a protocol two performance characteristics of each communication channel are relevant: its *throughput* and its *latency*.

The throughput of a channel is the amount of information that can be transmitted through the channel per unit of time. The amount of information can be measured in bits, bytes, or even "messages"; the latter is meaningful if all messages have (approximately) the same size. Thus, for example, we have that "bits per second" equals "bits per message" times "messages per second".

The latency of a channel is the amount of time that elapses between sending a message and arrival of the same message. In many cases latency is a more important parameter than throughput, particularly so when messages are small. As an example, in a simple (so-called) *hand-shake* protocol, where process X sends a signal to another process Y and then receives a signal as acknowledgement from Y, the delay experienced by X is twice the latency of the communication channel along which the signals are transmitted.

exercises

1. Every day, in an eight-hour flight, an airplane from Amsterdam to New York carries 100 boxes with 144 CD-ROMs per box; each CD-ROM contains 600 MByte of data. If we view this as a communication channel then what are its throughput – in megabits per second – and its latency?

2. A coaxial cable (at the bottom of the Atlantic Ocean) with a length of 7200 km connects Amsterdam and New York. Via this cable data can be transmitted at a rate of one bit per microsecond. Signals via this cable travel at the speed of light, which for our coaxial cable amounts to 144 000 km per second. What are the throughput – in megabits per second – and the latency (for a single bit) of this medium?

3. To transmit the 600 MByte contents of a single CD-ROM from Amsterdam to New York we can choose between either the airplane or the coaxial cable from the previous exercises. Which of the two delivers this 600 MByte contents fastest?

4. To transmit the 600 MByte contents of n CD-ROMs from Amsterdam to New York we can choose between either the airplane or the coaxial cable. What is the minimal value of n for which transmission by airplane is faster than via the cable?

2 Solutions

We present a few different solutions to the problem posed in the Introduction. We do so in an informal and sketchy way only. The purpose of this presentation is to give an impression of what makes distributed programs more difficult (than sequential ones) and of the variety in possible solutions. Formal developments of these solutions are certainly possible but require mastery of the appropriate formal techniques; fortunately, the problem is simple enough for an informal (and yet precise) treatment.

An important distinction is the one between solutions that are *fully distributed* and solutions that are not. A solution is fully distributed if the symmetry between the processes is retained and if no dedicated *centralized* processes – sometimes called *servers* – are introduced. Fully distributed solutions are generally preferred but in particular cases the use of servers may be very well justified.

We use N for the number of processes whose output streams have to be interleaved into the (one and only) input stream to be delivered to these processes.

2.0 A scheduling server

We introduce an additional process, called Server, that is connected with all N processes proper. A protocol meeting all requirements can now be formulated thus:

0. Each time a process produces an item in its output stream that process sends the item (in a message) to Server.

1. Server interleaves the items arriving from the N processes into a single stream; this is "the" input stream. This interleaving must be fair and order preserving; because Server may be a single, sequential process this is relatively easy.

2. Server sends each item added to the input stream to all N processes; thus, the input stream is distributed (and duplicated) over the processes.

This solution is simple and efficient; in addition, it has the following properties:

- The communication channels from the processes to Server must have the FIFO-property, otherwise preserval of the order of the items in the output streams cannot be guaranteed.
- The communication channels from Server to the processes must have the FIFO-property, otherwise preserval of the order of the items in the input stream cannot be guaranteed; as a result, it cannot even be guaranteed that all processes receive the same input stream.
- This solution is not fully distributed; actually, the mutual communication channels between the processes proper are not even used.
- Distribution of a single item to all N processes requires transmission of $N+1$ messages; because the N messages from Server to the processes can be transmitted concurrently, this yields a latency of 2 (message-transmission times) per item.
- High throughput: every process may send its next output items before earlier ones have been processed. Provided Server is fast enough the throughput of the communication channels determines the throughput of the system.

The required FIFO-properties can be obtained by means of additional (and independent) protocols, as suggested in exercise 0.

2.1 Mutual exclusion

Each process has its own copy of "the" input stream. To guarantee that these copies remain identical every item is added to all of them in one single indivisible action. To implement this indivisible action we use a *mutual exclusion* algorithm.

For the process producing output items this protocol can be formulated thus; here *enter* and *exit* denote the – yet to be designed – interaction with the mutual exclusion algorithm:

```
  enter
; send the item to each of the processes
; receive a signal from each of the processes
; exit
```

Each time a process receives (a message containing) an item it sends a signal – an empty message – to the sender of that item, so as to acknowledge inclusion of this item in its (copy of the) input stream:

```
on  receipt of an item  x  from process  p
→   add  x  to the local input stream
  ;  send a signal to  p
no
```

This action is likely to be implemented as a (concurrent) *co-process* of the process proper, because of the asynchronous nature of item arrival. In practical implementations this may take the shape of an *interrupt-service routine* or – similarly but more abstractly – an *event-handling procedure*.

The acknowledging signals are necessary to guarantee that the addition of the item to all input streams has been completed *before* the *exit*-operation of the mutual exclusion: thus, the addition to the input streams is implemented as an indivisible action.

Independently of which mutual exclusion algorithm is used, this solution has the following general properties:

- Because of the acknowledging signals the communication between a process and each of the co-processes effectively follows a hand-shake protocol. As a result, every communication channel always carries *at most* 1 item at a time. Hence, no FIFO-property is required.
- Because every channel always carries at most 1 item, and because the acknowledging signal must have arrived before the next item is sent, the effective throughput is twice the latency of the channel; hence, the effective throughput may be (much) lower than the channel's possible throughput.
- Distribution of a single item to all N processes requires transmission of $2 * N$ messages plus what is needed to achieve mutual exclusion. The latency is 1 plus what is needed for the *enter*-operation.

* * *

We now present two (of the very many possible) ways to implement mutual exclusion in a distributed collection of processes. Both can be viewed as a form of *token management*. The system contains a *single* token, it is a system invariant that, at any moment in time, at most 1 process "has the token", and

only a process that "has the token" is allowed to perform its critical actions. This guarantees mutual exclusion. For the sake of progress, of course, every process needing the toking must obtain it within a reasonable amount of time.

In this approach the operation *enter* amounts to "obtain the token" and *exit* amounts to "pass on the token". The two ways presented below differ in how the token is passed on.

2.1.0 a token server

In a state of rest, where no process needs the token, the token resides in a dedicated additional process called Server. A process needing the token acquires it from Server and later releases it to Server again. Thus, both *enter* and *exit* can be implemented by a simple hand-shake of signals:

enter: send a signal to Server
; receive a signal from Server

exit: send a signal to Server
; receive a signal from Server

The receipt of a (first) signal from Server carries the information that Server has granted the token to the process to which this signal is sent. Because *enter* and *exit* are always executed in strict alternation the signals need not carry additional information: the *first* signal from a process to Server represents a request and the *second* signal from the process to Server represents a release of the token.

Each time Server receives a request for the token it grants it to the requesting process by sending a signal to it, provided Server "has the token"; otherwise, this granting is *postponed* until Server has received the token back from the processes.

remark: It is a system invariant now that "Server has the token" is equivalent to "Server has received an even number of signals from every process". Similarly, "process p has the token" is equivalent to "process p has both sent and received an odd number of signals to and from Server".
□

This solution is about the simplest possible but it is not fully distributed. The implementation of *enter* adds 2 to the latency of item transmission.

2.1.1 distributed token passing

For the sake of progress the token must be passed on through the collection of processes, in such a way that every process regularly receives the token. A simple way to obtain this is to arrange the processes in a *ring* and have each process always send the token to its successor in the ring only. This guarantees that every process will receive the token again after it has propagated it to its successor, with a latency of N (signal times).

As before, propagating the token can be represented by an (empty) signal, as long as this causes no ambiguities with other uses of signals.

A process needing the token just waits until it arrives. A process *not needing* the token but receiving it anyhow may keep it "for a while" but eventually must propagate it, for the sake of those processes that may need it. As token arrival is an asynchronous event, a co-process is needed to take care of this propagation. Keeping the token "for a while" may help in reducing the amount of signals sent; thus, the choice of how long "for a while" will last reflects a compromise between the number of signals sent and the latency considered acceptable.

exercises

5. In the item-distribution protocol with mutual exclusion, signals are also used to acknowledge addition of an item to the input streams. Does or does not this use of signals interfere with the use of signals for token passing?

6. The above solution assumes the presence of exactly 1 token in the system. Initially, however, the system contains no tokens. Design a simple protocol to generate exactly 1 token.

2.2 A bakery algorithm

The position of an item in the input stream can be represented by its *ordinal number*, that is, the number of items (in the input stream) preceding that item. The order of the items in the input stream is fully determined by these ordinal numbers; hence, if these numbers can be assigned early enough, they can be used to construct the input streams for the N processes concurrently and yet in a unique way.

This gives rise to a protocol of the following structure: each time a process produces an output item the process assigns a unique ordinal number to the item and sends the item together with this ordinal number to each of the

processes. Upon receipt of a numbered item the receiving process keeps this item until the length of the local input stream equals the ordinal number of the item; only then is the item added to the input stream. Again, because item arrival is asynchronous, the implementation of this requires a co-process per process proper.

Apart from the (non-trivial) problem of how to assign the ordinal numbers, this protocol has attractive properties: the ordinal numbers identify the positions of the items in the input stream uniquely and, hence, no FIFO-properties are required of the communication channels: the items will be processed in the right order anyhow. Also, no acknowledging signals are needed and, as a result, a high throughput is possible (at least, as far as item transmission is concerned).

In this approach the only critical action is the assigment of an ordinal number to the items. The requirements are that to every item a unique number is assigned and that numbering is *consecutive*, which means that every natural number must actually be used. The latter is necessary for the sake of progress: otherwise, the construction of the input streams will come to a grinding halt (waiting for an ordinal number that will never appear).

For this purpose we introduce a single *shared variable* C whose value is a natural number; C represents the first number yet to be used as ordinal number. Hence, C's initial value is 0, and after every inspection of C's value, C is incremented by one, in one and the same indivisible action. Thus, for a process producing an output item x the protocol becomes:

$$\begin{array}{ll} & h := C \ ; \ C := C+1 \\ ; & \text{send } \langle x, h \rangle \text{ to each of the processes} \end{array}$$

Here $\langle x, h \rangle$ denotes the pair composed from x and h. Using a (local) variable l for the length of the local input stream and a (local) variable $\mathcal{B}$ containing the set of pairs that have been received but whose items have not yet been added to the input stream (because their ordinal numbers exceed l) we can formulate the protocol for reception of a numbered item as follows – again this must be implemented as a co-process –:

$$\begin{array}{ll} \mathsf{on} & \text{receipt of pair } \langle x, h \rangle \\ \rightarrow & \text{“add } \langle x, h \rangle \text{ to } \mathcal{B}\text{”} \\ \quad ; & \mathsf{do} \quad \text{“}\mathcal{B} \text{ contains a pair } \langle \cdot, l \rangle\text{”} \\ \quad & \quad \rightarrow \text{“select } y \text{ such that } \langle y, l \rangle \in \mathcal{B}\text{”} \ ; \ \text{“remove } \langle y, l \rangle \text{ from } \mathcal{B}\text{”} \\ \quad & \quad \ ; \ \text{“add item } y \text{ to the local input stream”} \ ; \ l := l+1 \\ \quad ; & \mathsf{od} \\ \mathsf{no} & \end{array}$$

* * *

We are left with the implementation of $h := C \; ; C := C+1$ as a single indivisible action. This requires a form of mutual exclusion again, but because this action now is the only one requiring mutual exclusion we can construct a dedicated, more "integrated", solution.

For example, we may introduce a process Server again, but now we let C be a local variable of Server; whenever Server receives a request it sends a response message containing the current value of C and then performs $C := C+1$ *before* it serves any new requests. As a result, a process needing a new ordinal number now has to perform a single hand-shake with Server only. (Server now corresponds to the ticket dispenser in a bakery!)

More interesting, and fully distributed, is the integration with token passing. The crucial observation is that only the process that "has the token" may inspect and change C, thus guaranteeing the required indivisibility. Hence, although C is a shared variable, only the process having the token is accessing it and, therefore, C must only be present in that process. This can be implemented in a very simple way: the value of C is just passed on with the token!

Eindhoven, 21 march 2001

Rob R. Hoogerwoord
department of mathematics and computing science
Eindhoven University of Technology
postbus 513
5600 MB Eindhoven