UDC 519.6

A CUSTOMIZABLE VERSION OF THE TERMINATION DETECTION ALGORITHM FOR INDIGO FRAMEWORK

V. Kolesnikov

Sumy State University,
Rymskogo-Korsakova str., 2, Sumy, 40007,
e-mail: *v.kolesnikov@cs.sumdu.edu.ua*

This paper presents a customizable version of the termination detection algorithm for distributed systems. We follow an approach developed for designing general purpose distributed algorithms customizable to a specific operational context within the InDiGO framework [1]. To customize such algorithms, they must be expressed in a form amenable to customization. We present a mechanism which allows a designer to expose the design knowledge related to the communication structure of an algorithm. This involves identifying interaction sets used for communication in an algorithm, and defining the semantics of these sets in terms of queries supported by the analysis infrastructure of the InDiGO framework. The benefits of customizable versions of distributed algorithms are presented using an example of a customizable version of the termination detection algorithm.

Key words: distributed algorithms, termination detection algorithm, customization, InDiGO, frameworks.

1. Introduction

In the traditional approach for the development of distributed applications, the application uses the basic event service to implement interactions between its components [2–4]. In general, an application may require a richer set of distributed system services. For example, in bidding applications, we may need a termination detection algorithm to detect when the bidding is over. To isolate the designer from the intricacies of a distributed system, one can provide a library of distributed algorithms, which implements different types of services.

When an approach to provide a library of distributed algorithms is used, the designers of the library algorithms are faced with two opposing forces. One is to develop generic reusable algorithms that can be used in a wide variety of applications. On the other hand, applications may require algorithms to meet stringent performance constraints, which may force the designer to develop customized versions of the algorithms. The following example illustrates this tradeoff:

Distributed algorithms often do not make any assumption regarding the application and are therefore conservative in nature. However, application components may follow a specific communication pattern or topology. For example, in a tele-teaching application, sessions or groups may be formed for different purposes with varying number of participants. Each such group may follow a specific communication pattern (for example, an answer message may be sent only in response to a question message) or a topology (for example, a ring or a star), which the underlying algorithms may be able to exploit. Thus, a straightforward use of a generic algorithm may not be efficient and this may force the designers to come up with their own implementations.

As an example of a possible optimization, let's consider a termination detection problem. In general, an algorithm for termination detection has to determine that all components are passive and all channels are empty. In a particular application, however, the

© Kolesnikov V., 2021

passive states of the components may be dependent on each other. As a simple example, if component A communicates only with B and performs tasks assigned by B only, then A will always be passive whenever B is passive. Such dependencies can be used to reduce the number of components to be polled for passive states.

For example, for a star topology shown in Fig. 1, if process 2 would like to determine termination, it would send a marker message to processes 1, 3 and 4.

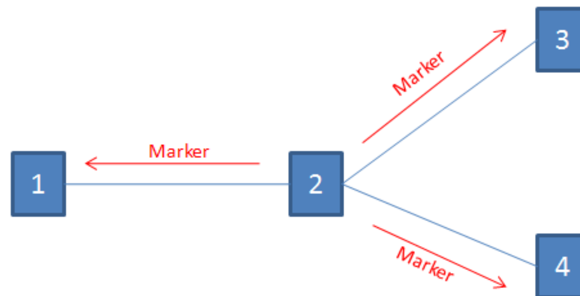

Fig. 1. Termination detection example – sending of Marker messages

The processes respond with either Done message (if they are passive) or Continue message (if they are still active) as in Fig. 2. If process 2 receives Done message from all the processes and it has remained passive since it sent the marker messages out, termination is detected.

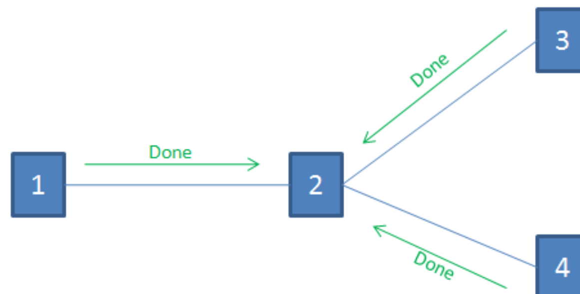

Fig. 2. Termination detection example – receiving of Done messages

But, in a particular application, if component 1 communicates only with component 2 and performs tasks assigned by 2 only, then 1 will always be passive whenever 2 is passive.

For example, Answer message could only be sent in response to a Question message (see Fig. 3). Then, if process 2 receives all the answer messages from process 1 and is passive, then process 1 is passive also (since it can only be activated by a message from process 2).

Then, in generic algorithm, process 2 will still send a marker message to all processes. But the message to process 1 is not needed (see Fig. 4).

Message Done from process 1 is not needed either (see Fig. 5). So, the generic algorithm is going to be inefficient. Such dependencies as described above can be used to reduce the number of components to be polled for passive states.

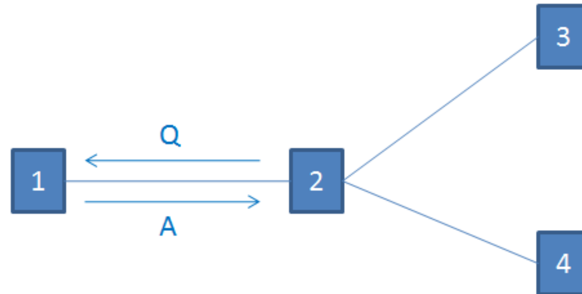

Fig. 3. Termination detection example – ordering on Q/A messages

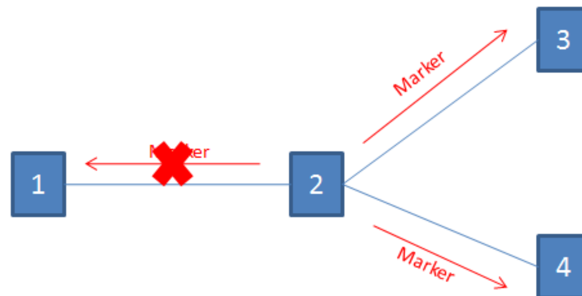

Fig. 4. Termination detection example – taking ordering information into account for Marker messages

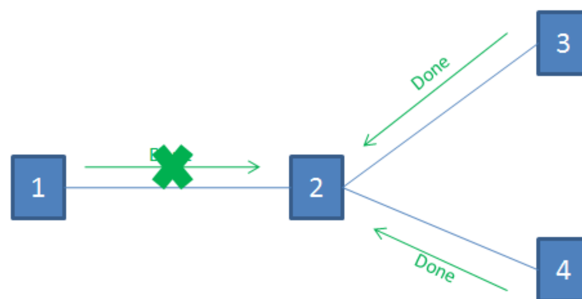

Fig. 5. Termination detection example – taking ordering information into account for Done messages

One can take advantage of the application structure to optimize the performance of the distributed algorithms. So, if the algorithms in the library are used as-is, the resulting implementations may be inefficient. In such cases, an application developer may be tempted to develop algorithms suited to the application from scratch.

2. Termination detection algorithm

2.1. The Termination Detection Problem

The problem of detecting that a distributed algorithm has terminated is both important and non-trivial. Even if observation has shown that all the constituent processes of the algorithm are in a passive state – that is, are not active – this cannot be taken as a proof that the algorithm as a whole has terminated because the process observed to be passive maybe reactivated by a message from a process that has not yet been observed and which then becomes passive. The problem would be simple if knowledge were available, at any instant, of a global state that took into account both the processes and the communication channels. Designing an algorithm for the problem thus comes down to designing a distributed control mechanism that will recognize a particular state of global stability, that of termination.

A process is said to be active if it is executing the instructions of its program and is passive if it is in any other state. A passive process can be either terminated, having completed its task, or waiting for messages from other processes. If all the processes are passive and no messages are in transit, the complete distributed algorithm is said to be terminated.

2.2. Description of a Distributed Termination Detection Algorithm

We present the distributed termination detection algorithm that was first described in [5]. The algorithm is shown in Fig. 6 and works as following. Processes are labeled $P_i, 0 \leq i \leq n$. We employ a *token* to transmit the values $quiet_p$. The token cycles through the processes visiting $P_{(i+1) mod\ n}$ after departing from P_i. A cycle is initiated by a process P_{init}, called the initiator. If the token completes the cycle (returns to P_{init}) after visiting all the processes and if all processes P return a value $quiet_P$ of *true* in this cycle, then the inititator detects termination, i.e. it sets *claim* to *true*. If any process q returns a value $quiet_q$ of *false* in a cycle, then the current cycle is terminated and a new cycle is initiated with q as the initiator. A process ends one observation period and immediately starts the next observation period when it sends the token. The algorithm, described next in detail, shows how $quiet_P$ is set.

There are no shared variables in a distributed system. However, for purposes of exposition we assume that *claim* is a shared global variable which has an initial value of *false* and which may be set *true* by any process. Such a global variable can be simulated by message transmissions; for instance, the process that sets *claim* to *true* may send messages to all other processes notifying them.

Two types of messages are employed in the termination detection algorithm:

- *(marker)*
- *(token, initiator)*

Each process has the following constants and variables that will be subscripted, by i,

when referring to a specific process *i*.

- *ic*: number of incoming channels to the process, a constant
- *idle*: process is idle
- *quiet*: process has been continuously idle since the token was last sent by the process; *false* if the token has never been sent by this process
- *have_token*: process holds the token
- *init*: the value of the initiator in the *(token, initiator)* message last sent or recieved; undefined if the process has never received such a message
- *m*: number of markers received, since the token was last sent by the process

Initial conditions

- The token is at P_k
- m_i = the number of channels from processes with indices greater than i, for all i, i.e., the cardinality of the set, $\{c \mid c \text{ is a channel from } P_j \text{ to } P_i \wedge j > i\}$
  (This initial condition is required because otherwise, the token will permanently stay at one process)
- $quiet_i = false$, for all i
- $have_token_i = true$ for $i = 0$; *false* otherwise
- $init_i$ is arbitrary, for all i

Code for process P_i:

```
01 :: receive(marker)
02     m ← m + 1
03 :: receive(app)
04     if (quiet)
05         quiet ← false
06 :: receive(token, initiator)
07     init ← initiator
08     have_token ← true
09 :: (have_token ∧ (ic = m) ∧ idle)
10     if (quiet ∧ (init = i))
11         claim ← true // termination detected
12     else if (quiet ∧ (init ≠ i)) // continue old cycle
13         m ← 0
14         send marker to all neighbors
15         have_token ← false
16         send(token, init) to P_(i+1)mod n
17     else if (!quiet) // initiate new cycle
18         m ← 0
19         quiet ← true
20         init ← i
21         send marker to all neighbors
22         have_token ← false
23         send(token, init) to P_(i+1)mod n
```

Fig. 6. Distributed termination detection algorithm for an arbitrary topology

3. Development of customizable algorithms

In this section, we discuss the design of customizable distributed algorithms. One can follow several complimentary approaches to build a customizable distributed algorithms library. One approach is to develop a *set of algorithms* for the same problem, with each algorithm offering advantages over its alternatives in specific operational contexts. For example, this approach was followed in [6] to design a set of mechanisms for event communication whose relative performance are dependent on factors such as number and location of producers and consumers and publication rates. In order to analyze the application, tools were developed to select the most appropriate mechanism for each event type. In this paper, we follow a complimentary approach wherein we want to customize specific algorithms themselves (rather than selecting between algorithms). To enable customization, an algorithm developer must expose design knowledge pertaining to an algorithm in a form which can be leveraged for analysis [7, 8]. Algorithms have been designed with parameters such as maximum number of possible node failures or conflict relations to adapt their behavior. While parameters such as conflict relation exploit application semantics, they do not directly analyze the application structure for optimization.

In this paper, we study mechanisms to expose knowledge related to the communication structure of an algorithm for possible optimizations.

3.1. Interaction sets

We require the algorithm designers to adopt the following approach:

For each algorithm *alg*, the designer first identifies the *interaction sets*, denoted by *alg.interaction_set*, which characterize its communication structure and specify the processes participating in each interaction. The designer then writes *alg* in terms of these sets. As we will show later, this involves a simple transformation of the existing algorithms.

Alternatively, one can define sets with well defined meanings for a class of algorithms. The sets could be general enough to be used in a number of distributed algorithms or could be very specific to a particular type of algorithms or even a particular algorithm.

In a later part of this paper we look at an example of a distributed algorithm for which we define algorithm specific interaction sets that describe the communication structure of that particular algorithms.

3.2. Membership criteria for interaction sets

Next, for each interaction set *interaction_set*, the algorithm designer defines the *membership criterion*, denoted by *interaction_set.membership_criterion*, which specifies the criterion for a process to be involved in an interaction and so defines if a process is a part of the interaction set. The membership criteria must be defined in terms of the queries supported by the analysis infrastructure. This criterion is a problem-specific property that a process must satisfy to participate in the interaction. As shown in the example below, this allows the sets to be defined in a problem-specific manner (rather than only in terms of physical topology).

The membership criteria need to be expressed in such a form that tools are able to

parse it. For that reason, we require the algorithm developer to specify membership criteria for interaction sets in a standardized form. An example of such an input file for the membership criterion of an interaction set is shown in listing 1. It specifies the name of the interaction set, the name of the query to run and the arguments for the query in a comma delimited format.

```
1 SRT.i , ALL, not exclusive , i.in_cs = 1, j.in_cs = 1
2 ...
```

Listing 1. Sample of membership criteria

InDiGo framework supports a number of basic queries that can be used in defining membership criteria.

It is the responsibility of the algorithm designer to ensure the correctness of the algorithm written in terms of interaction sests with respect to membership criteria. Just like in traditinal algorithm development, algorithm designers are to provide correctness proofs with respect to algorithm properties.

3.3. Rules for dynamic updates to the interaction sets

Finally, we allow the algorithm designer to leverage the design knowledge and provide information for *dynamic updates* to the interaction sets.

In an algorithm, as a result of message passing, a process may obtain knowledge of the states of the application entities at other processes. For example, when process i receives a request message from process j in a mutual exclusion algorithm, it knows that an application entity at process j is in the requesting state. This information can be used to further constrain the interaction sets via dynamic update.

As with the membership criteria for interaction sets, information on dynamic updates to the interaction sets needs to be expressed in such a form that tools are able to parse it. For that reason, we also require the algorithm developer to specify information on dynamic updates to the interaction sets in a standardized form. We require the algorithm developer to add a condition to the input file. This condition will describe which node in the abstraction model of our system will be enabled when a process is in a certain state.

These dynamic rules are used during the system execution to further constrain interaction sets based on the information received through message passing of the executing system.

3.4. Customized Version of the Distributed Termination Detection Algorithm

The communication structure of the algorithm shown in Fig. 6 can be characterized by the following three interaction sets:

- *send_marker_to* (SMT) is the set of all neighbor processors to which a marker message has to be sent.
- *wait_response_from* (WRF) is the set of all neighbor processors from whom a marker message is to be received.
- *send_token_to* (STT) is a singleton set consisting of the id of the next processor to send token to.

The algorithm written using these sets is shown in Fig. 7. As can be seen, the transformation is simple.

```
    Code for process P_i:
01  :: receive(marker)
02      m ← m + 1
03  :: receive(app)
04      if (quiet)
05          quiet ← false
06  :: receive(token, initiator)
07      init ← initiator
08      have_token ← true
09  :: (have_token ∧ (WRF.size = m) ∧ idle)
10      if (quiet ∧ (init = i))
11          claim ← true // termination detected
12      else if (quiet ∧ (init ≠ i)) // continue old cycle
13          m ← 0
14          update_SMT()
15          send marker to SMT
16          have_token ← false
17          update_STT()
18          send(token, init) to STT
19      else if (!quiet) // initiate new cycle
20          m ← 0
21          quiet ← true
22          init ← i
23          update_SMT()
24          send marker to SMT
25          have_token ← false
26          update_STT()
27          send(token, init) to STT
```

Fig. 7. Customized version of distributed termination detection algorithm

Next, we define the *membership criteria* for these sets. For simplicity, we have assumed that at most one application component is mapped to each site and will use C_i to denote the component mapped to site i. Let Nbr_i denote the set of processes such that a component at processor i communicates with a component at processor j in *App*. We define SMT_i as following: $SMT_i = \{j : j \in Nbr_i)\}$. SMT set for processor i specifies neighbours of i. WRF_i is defined the same as SMT_i.

We define $STT(i)$ as following:

$$STT(i) = \begin{array}{ll} \text{if } i < n-1 & \\ & j = i+1 \\ \text{else} & \\ & j = 0 \end{array}$$

This specifies that i must send the token to $P_{(i+1) mod\ n}$ processor next. For dynamic membership, we identify assertion "$enabled(C_i.passive_noact)$", stating that component C_i is passive and has not sent any messages to activate other components since its own last activation. Calls to procedures $update_SMT$ (lines 14 and 24) and $update_STT$ (lines 17 and 26) are added and called before the sets are used. The code for these procedures is synthesized by the dynamic optimization rules.

4. Conclusion

In this paper we presented an approach to designing general purpose distributed algorithms customizable to a specific operational context. As an example, we used a termination detection algorithm. We have developed a mechanism, which allows a designer to expose design knowledge related to the communication structure of an algorithm. This involves identifying the interaction sets used for communication in an algorithm, and defining the semantics of these sets in terms of queries supported by the analysis infrastructure of the InDiGO framework.

References

1. Kolesnikov V. InDiGO: An infrastructure for optimization of distributed algorithms / V. Kolesnikov, G. Singh,– 7th International Symposium on Parallel and Distributed Computing, 2008. – P. 401–408.
2. Birman K. Reliable Distributed Computing with the ISIS toolkit / K. Birman, R. Renesse. – IEEE Computer Society Press, 1994.
3. Guerraoui R. Consensus service: A modular approach for building fault-tolerant agreement protocols in distributed systems / R. Guerraoui, A. Schiper. – IEEE International Symposium on Faul-Tolerant Computing Systems, 1996.
4. Kalantar M. Causally ordered multicast: the conservative approach / M. Kalantar, K. Birman. – IEEE Int'l Conference on Distributed Computing Systems, 1999.
5. Chandy K. A paradigm for detecting quiescent properties in distributed computations / K. Chandy, J. Misra. – Springer-Verlag New-York, 1985. – P. 325–341.
6. Singh G. Configurable event communication in cadena / G. Singh, P. Kumar, Q. Zeng. – IEEE Conference on Real-time Applications and Systems, 2004.
7. Singh G. Methodologies for optimization of distributed algorithms and middleware / G. Singh, V. Kolesnikov, S. Das. – 22nd IEEE International Parallel and Distributed Processing Symposium IPDPS, 2008.
8. Kolesnikov V. Methodologies for optimization of distributed algorithms and middleware /V. Kolesnikov. – 9th International Symposium on Parallel and Distributed Computing, ISPDC 2010. – P. 85–92.

Article: received 20.10.2021
revised 10.11.2021
printing adoption 24.11.2021

ВЕРСІЯ АЛГОРИТМУ ВИЯВЛЕННЯ ЗАВЕРШЕННЯ, ЩО ПРИДАТНА ДО НАЛАШТУВАННЯ У ФРЕЙМВОРКУ INDIGO

В. Колесніков

Сумський державний університет,
вул. Римського-Корсакова, 2, Суми, 40007,
e-mail: *v.kolesnikov@cs.sumdu.edu.ua*

Подана придатна до налаштування версія алгоритму виявлення завершення у розподілених системах. Ми дотримуємось розробленого підходу для проєктування розподілених алгоритмів загального призначення, налаштованих під конкретний операційний контекст у рамках фреймворку InDiGO [1]. Для цього такі алгоритми мають бути виражені у формі, що підлягає налаштуванню. Подаємо механізм, який дає змогу дизайнеру виявити знання з дизайну, пов'язані з комунікаційною структурою алгоритму. Це передбачає виявлення наборів взаємодії, що використовуються для спілкування в алгоритмі, визначення їхньої семантики з погляду запитів, підтримуваних інфраструктурою аналізу фреймворку InDiGO. Переваги налаштованих версій розподілених алгоритмів представлені на прикладі кастомізованої версії алгоритму виявлення завершення роботи у розподілених системах.

Ключові слова: розподілені алгоритми, алгоритми виявлення завершення роботи у розподілених системах, налаштування, InDiGO, фреймворки.