Chapter 19
A Java-Based Distributed Approach for Generating Large-Scale Social Network Graphs

Vlad Şerbănescu, Keyvan Azadbakht and Frank de Boer

19.1 Introduction

Distributed systems and applications require large amounts of resources in terms of memory and computing power and are becoming a standard for large businesses and enterprises [12] within and outside the domain of Computer Science. A very important topic for distributed applications is Big Data management and more specifically the generation of large-scale social networks graphs where the number of nodes reaches very large numbers. Analysis of such networks is of importance in many areas, e.g., data mining, network sciences, physics, and social sciences [3]. The need for efficient and scalable methods of network generation is frequently mentioned in the literature [8], particularly for the preferential attachment process [1, 13, 14]. Barabasi–Albert model, which is based on preferential attachment (PA) [4], is one of the most commonly used models to produce artificial networks, because of its explanatory power, conceptual simplicity, and interesting mathematical properties [13]. Nevertheless the large number of nodes in such graphs may not fit in the memory on one machine. The need for efficient solutions which provide scalability also requires more computational resources as well as implementation considerations. As such, distribution and synchronization are two main challenges. In this chapter, we investigate as a case study a distributed solution for PA-based graph generation which avoids low level synchronization management, thanks to the notion of cooperative scheduling and futures.

V. Şerbănescu (✉) · K. Azadbakht · F. de Boer
Centrum Wiskunde and Informatica, Science Park 123, 1098 Amsterdam, XG, Netherlands
e-mail: vlad.serbanescu@cwi.nl

K. Azadbakht
e-mail: kazadbakht@cwi.nl

F. de Boer
e-mail: frb@cwi.nl

F. Pop et al. (eds.), *Resource Management for Big Data Platforms*,
Computer Communications and Networks, DOI 10.1007/978-3-319-44881-7_19

In several examples of distributed applications such as high-energy physics applications, research digital libraries or secure banking systems with millions of users, communication between remote machines is a significant challenge. Modeling distributed communication such that it can be analyzed at development phase for synchronization issues or deadlocks allows the design of reliable and efficient software that will not require extensive testing and debugging. We provide in this paper a library that allows a more intuitive mapping from a modeling language, the abstract behavioral specification language(ABS) [6], to a programming language. Since the modeling language is tailored toward asynchronous communication, the distributed implementation of the library will not require any remote object field access or synchronous remote method invocations.

The expressive power of the above-mentioned library is shown in the PA-based graph generation case study where the array representing the graph is partitioned and located on remote machines, each is owned by an actor. According to PA, the slots of each array are resolved by the values from (the same or) other arrays by which the new connections between the nodes are formed. The *future* construct provides a means for each actor on a machine to synchronize on the return value from a process (i.e., a runtime method execution of an actor) on remote machines. The process itself can also suspend on a boolean condition (i.e., the continuation of the process can be activated when the condition is evaluated to True), and then returning the value. As shown in the case study, using such powerful constructs eliminates the need for low level synchronizatoin mechanisms.

While the Java language is one of the most popular, intuitive and easy to use languages in terms of software development and has significant support for parallel and distributed programming, its most basic entity for representing a block of running code is a Thread which is very expensive memory-wise. There are several interfaces which abstract lightweight tasks and subtasks, but in order to schedule and preempt these tasks they still need to be encapsulated in the form of a Thread. The issue appears when an application requires multiple context switches between several tasks which have significant call stack sizes. As this number becomes very large the thread explosion affects the main memory thus the application's performance. The major contribution of this chapter is to provide a library with distributed support of the actor-based model in the Java language, with enhanced future synchronization capabilities that become available in a distributed setting. Furthermore we introduce data structures to manage propagation of futures and allow them to be garbage collected on each machine. We also provide a notification mechanism for future resolution in order to avoid inefficient busy-waiting loops. The rest of this paper discusses related work in Sect. 19.2, followed by a detailed explanation of the new library in Sect. 19.3. We present a case study that is suitable for evaluating our solution in Sect. 19.4 and its high-level implementation in Sect. 19.5. We draw the conclusions and present the next steps for this work in Sect. 19.6.

19.2 Background and Related Work

In the ABS, the core language semantics imposes that all objects created in the program are actors with an independent behavior, with a possibility to communicate with other actors and use futures to synchronize at certain execution points. The language groups actors into Concurrent Object Groups (COG) which allow objects to communicate with each other synchronously, but apart from the actors in the same COG, all other actors are considered remote and may only invoke each other asynchronously. The physical location of an actor in ABS is completely transparent as there are no virtual machines or IP addresses inserted in this high-level language. The language features one construct for the asynchronous communication, another construct for blocking an actor's execution on a future and a third and very powerful construct for suspending and scheduling methods within one single actor, a construct that introduces the notion of cooperative scheduling.

These two latter constructs can be preceded by annotations that allow custom schedulers to be defined in order to satisfy an application's specific requirements. Further annotations can be associated to method calls to specify costs and deadlines in order to create a very powerful scheduler. All these constructs are written in a very simple and concise way in ABS, in order to allow system designers a simple view of their application which can be even large enough to be deployed in a cloud environment [7]. However, from this modeling language we need to generate code for programs to execute on multiple resources, tasks to be submitted to those resources and also incorporate the powerful schedulers. To this end ABS has several execution backends in a simulation language (Maude), functional language (Haskell), and object-oriented languages (Erlang and Java). Our focus for this contribution is the Java language backend for which the translation process of distributed applications may create multiple redundant objects, threads, and data structures that significantly impact performance.

Listing 1 Scheduling in ABS example

```
interface Ainterface {
    Int recursive_m(Int i);
}

interface Binterface{
    Int  compute( );
}

class A implements Ainterface{

    Int recursive_m(Int i){
        if (i>0){
            this.recursive_m(i – 1);
        }
        else{
            B computation = new B ( );
            Future f = computation ! compute( );
            await f ?
}   }   }

class B implements Binterface {

    Int compute( ){
        Int result;
```

```
        /*do some work */
        return result;
}   }

{ // Main block:
    Int i = 0;
    A master_i = new A ( );
    List futures = EmptyList;

    while (i < 100){
        Future f = A ! recursive_m (10);
        futures = cons( f, futures );
    }

    while ( futures != EmptyList ){
        Future f = head (futures);
        futures = tail (futures);
        f.get ;
}   }
```

To illustrate how cooperative scheduling works, we look at a simple program in Listing 1. The program creates an object of Class A which contains method "recursive_m." The construct "o.recursive_m" is a regular synchronous call that must be executed without any preemption. Inside the method, we create an object of class B which has a method "compute." The construct "computation!compute( )" is an asynchronous call that allows the object of class B to execute in parallel the method "compute." At this point there are two constructs for synchronizing with a call that executes in parallel with the current object. The first construct, "f.get" is at the level of the object and forces the current object to block and wait for the completion of method "compute" that was captured in the future f. The second construct "await f ?" is more fine grained in the sense that only the current method that is executing this statement, namely "computation!compute( )" blocks, while subsequent calls of "computation!compute( )" resulting from the main for loop, can be scheduled and run by the object o. An important observation here is to understand that all calls like "computation!compute( )" are inserted into a queue that each object has and are scheduled to be run by the same object and not in parallel. The degree of parallelism is determined by the number of objects created.

The ABS-API library [9–11] was introduced as a solution to translate ABS code into production code initially for parallel applications. Java 8 new features allow wrapping of method calls into lightweight lambda expressions such that they can be put into a scheduling queue of an Executor Service to which the running objects are mapped, significantly reducing the number of idle Threads at runtime. Furthermore we minimized the number of threads created by saving the call stack of suspended methods within an actor caused by the "await" statement. Our first solution to achieve this was to add a central context for all actors in the system and follow this execution sequence.

- Each asynchronous call/invocation is a message delivered in the corresponding object's queue.
- All objects in the same Concurrent Object Group (COG) compete for one Thread.
- A *Sweeper Thread* decides which task should be created and be available for execution from the available queues.
- A thread pool executes available tasks based on a work stealing mechanism.

- On every await statement, we try to optimally suspend the message thread until the continuation of the call is released.

This *Sweeper Thread* however becomes a bottleneck when the number of actors is very large while also making actors dependent on each other. The new library that is the main contribution of this chapter, however, is tailored to support distributed actor-based programming and therefore requires a different organization of thread management and future propagation. Furthermore in a distributed setting there can be no centralized thread for all actors, therefore we propose a new solution that replaces the purpose of this thread.

19.3 Description of the Distributed ABS-API Library

In this section we describe the newest version of the ABS-API library that is written in Java to support an actor programming model in a distributed setting, with enhanced cooperative scheduling, distributed future control and garbage collection. We present a whole new and simple format for classifying actors based on their intended behavior. In this version we optimize even further the memory management of actor by reducing the number of Threads created at runtime. We use certain triggers to determine the start and ending of a live context and eliminate the use of redundant Threads that correspond to the running process of an actor. We introduce a class hierarchy of actors running on remote hosts, on local host and actors whose functionality is reachable from a remote host. This hierarchy simplifies garbage collection and reduces the number of peer-to-peer communications between remote hosts, as well as offering a clear separation between an application running on a single machine or in a distributed environment.

19.3.1 Actor Class Hierarchy, Naming Scheme and Asynchronous Communication

In the previous implementation of the ABS-API we had one single interface to allow an actor programming paradigm. This single interface encapsulated the entire behavior of the actor that comprised of the continuous running cycle, the task message queue, the single thread restriction and the cooperative scheduling of its suspended tasks. However, when deploying actors in a distributed environment they have particular locality and visibility characteristics. It is also important to take into account how actors will communicate with each other depending on their location and what elements need to be specific to the machine. Therefore an actor needs to have a global identity (represented by a Java URI) making it discoverable on all the machines of the application while its internal structure exists on only one machine. The URI takes the format of "IP:actorName", where IP is the host machine's address and *actorName* is

a unique identifier that distinguishes between actors on this machine. This allows for a scheme where local generation of actors avoids inconsistencies at a global level. The discovery mechanism is very simple:

- An actor is instantiated on only one machine and is given a URI comprised of the machine's IP and a unique name.
- To communicate with a remote actor, a machine requires a reference with the unique global identifier.

For inter-machine communication each two machines in the system are connected by one socket and actors send messages through the machine's socket streams. Furthermore, each machine maintains a *actorMap* of all of its actors in order to have a mapping between the Java URI and the Java reference of the actor such that it can forward remote requests to the correct actor. A certain special type of actor is introduced that is classified as local and has no global identity such that it is not reachable by other machines and can only receive messages from other actors on the machine it runs on. These particular characteristics allow for a simple classification of the actors according to the class diagrams in Fig. 19.1 and provide a clear separation between a parallel and a distributed setting.

The diagram in Fig. 19.1, presents how we classify actors based strictly on the machine on which they reside and therefore their physical existence on the machine. The API has a parent abstract class called *DeploymentComponent* which maintains data specific to the machine. Firstly, it contains a customized ThreadPoolExecutor to which the actors residing on the machine will submit their tasks. This ThreadPool

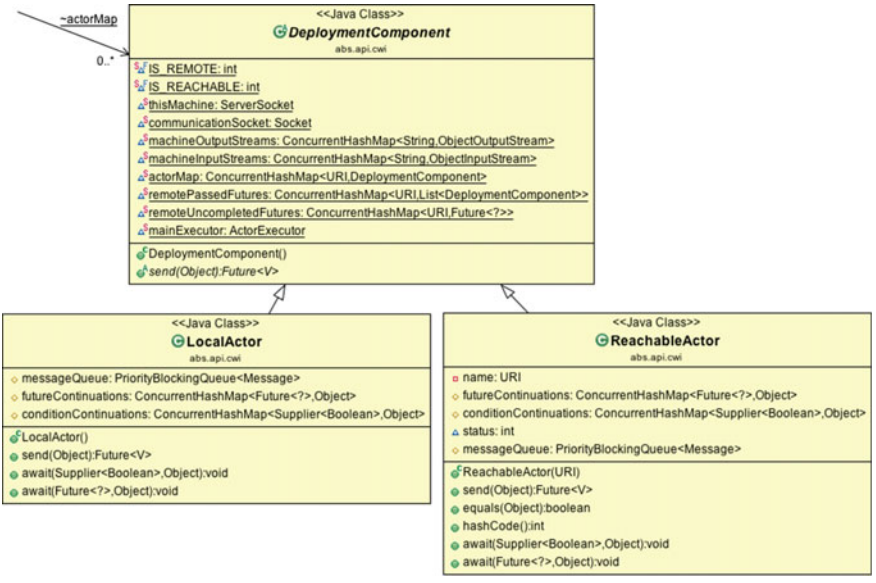

Fig. 19.1 Class diagram based on ABS transparency

Executor is available to all actors on one machine and it has an overridden *afterExecute* method to control several behaviors specific to actors which we will discuss further in this section. Secondly, the class also contains the *actorMap* of all the actors that are initialized on one machine such that remote messages can be routed to the correct actor. Finally, the class contains a table of the socket streams with all other machines in the system that grows and shrinks dynamically, as more machines are added to the system. An important observation is to notice that socket streams are initialized only when a remote actor belonging to a node that was previously unknown is instantiated in the system and a *listener* thread is assigned to the stream processing either incoming messages to the machine and as such, only if the setting is distributed. The machines communicate through serialized messages and objects that can be of four types that will be explained in the next two subsections:

- asynchronous method invocations.
- futures that are passed as parameters.
- a resolved future result.
- actor URI or futureID used to identify actors and futures on remote machines.

The DeploymentComponent class is subclassed into a *LocalActor* class which represents the simplest abstraction of an actor that is not connected to the outside of the machine. This actor has a message queue that can receive asynchronous messages that are executed in a FIFO order with submissions to the machine's thread pool. An important optimization is introduced in the execution instance of a standard actor. Instead of spawning a task that continuously loops through an actor's queue from the point it is instantiated, the task is now spawned only when the first message is introduced in the queue and finishes once the queue is empty. Therefore actors no longer have a live thread corresponding to their run if they are idle. The second subclass of the *DeploymentComponent* is the *ReachableActor* which has two very distinct behaviors, but ensures the transparency of the ABS language presented in Sect. 19.2. This class can identify either an actor that is extended with distributed support to receive remote calls or a remote actor which forwards messages to the correct machine. In both cases the actor is instantiated with the global identifier that we discussed earlier in this section, and this identifier distinguishes its general behavior on the machine. If its identifier's host IP is the same with the machine IP, then it behaves like a standard actor, only it is included in the machine's *actorMap* such that incoming messages can be forwarded to it for execution. On the other hand, if the IP differs from the machine IP, the actor is a remote actor and it is only a reference used for transparency on the machine. In this case, asynchronous messages are forwarded to the machine's output stream such that they can be sent via the machine socket to the machine where the remote actor actually resides.

19.3.2 Distributed Futures Control

The most important feature of our library is that it now has support for programming with distributed actors. A more detailed illustration in Fig. 19.2 explains the role of the *ReachableActor* on both a local and remote machine. In this setting we have an actor *a1* with a unique global identifier which is a Java URI that is "IP2:a1", where IP2 is the IP address of Node 2 on which the actor was instantiated and a1 is a unique identifier of the actor. Node 1 has a reference to this actor and its interface which contains method *m()* is also available. An important objective of our solution is to avoid actors entering a busy-waiting loop in order to check the resolution of futures. To achieve this, we insert a *remoteUncompletedFutures* data structure which retains all the futures that were generated by calls to remote actors. The machine then sends *a serialized lambda expression* of the asynchronous method call to the socket. Each machine is aware of the senders of incoming messages, therefore when an actor completes a remote call, the *serialized result* of the actor can be sent back as a reply. This behavior is part of the *afterExecute* method of the machine's main executor and

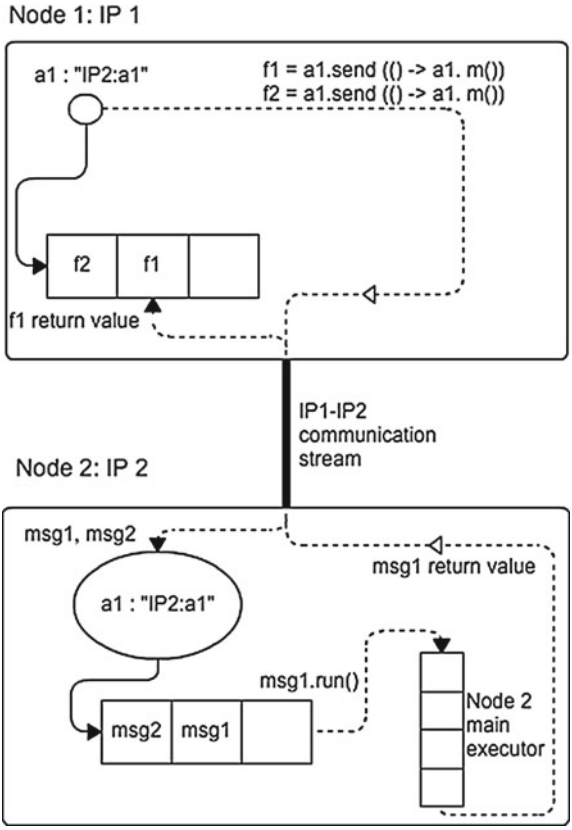

Fig. 19.2 Future Flow

is illustrated by the state *afterExecture* in the state diagram of the actor Fig. 19.4. In Sect. 19.2 we discussed how messages in an actor are executed in the order that they arrive in its queue and how the *await* statement allows for the rescheduling of these messages. To allow remote actors to identify which reply belongs to which future in the queue we introduce a naming scheme in the form of "IP:f" where IP is the address of the actor that will complete the future and f the unique global identifier (futureID) of the future.

The general mechanism is best described in terms of an example scenario with two asynchronous calls to the same actor:

1. Node 1 sends the following sequence of messages to actor *a1* on Node 2.
   - A futureID "IP2:f1" identifying the future that will be generated by the following asynchronous method call.
   - A pair $< IP2 : a1, m() >$ representing the first asynchronous method call to actor *a1*.
   - A futureID "IP2:f2" identifying the future that will be generated by the next asynchronous method call.
   - A pair $< IP2 : a1, m() >$ representing the second asynchronous method call to actor *a1*.
2. The two uncompleted futures *f1* and *f2* and their corresponding identifiers are stored as mappings as *remoteUncompletedFutures*.
3. Actor *a1* receives from the socket stream the two identifiers and two messages *msg1* and *msg2* in the same order and inserts them in the message queue.
4. Actor *a1* schedules *msg1* and *msg2* in a FIFO order on Node 2 main executor unless rescheduled by an *await* statement.
5. When either message has finished executing, the *afterExecute* method of the main executor sends back the corresponding futureID within either pair $< IP2 : f1, result >$ or $< IP2 : f2, result >$back to the socket stream where the message came from.
6. The socket stream forwards the result to Node 1.
7. Future *f1* or *f2* is completed with the received result depending on the futureID.

The semantics of ABS allows actor references and futures to be passed remotely through asynchronous method calls. However the semantics restrict actors from accessing fields of remote references or making synchronous calls on these references. The transparency feature of ABS means that remote futures are accessible by any actor and can be used together with the *await* and *get* statements to synchronize. A more difficult challenge is how futures are propagated throughout the system as parameters of method calls and when they become available for garbage collection on each machine. While remote objects that may be passed as parameters are handled by the class hierarchy, remote futures need a heuristic to be propagated and notified of completion once they are passed as parameters. A *serialized future* object, together with the futureID, needs then to be sent before the actual method call that contains it, such that it can be identified on the remote machine. This is a different

type of message from the one that just sends a futureID like in the previous scenario, as the remote actor actually needs the object to call *get* and *await* statements on. This future is then inserted into a table of *remotePassedFutures* and the corresponding list of machines to which they have been passed as parameters, or the table is simply updated with another machine if the future already exists. Whenever an actor passes a future as a parameter of a remote asynchronous class it takes the following steps:

1. It checks if the future is completed and if so, sends it via the socket stream before sending the asynchronous call.
2. If it is uncompleted, the future is still sent before the call, but also saved in a map with the format $< futureID : List < DeploymentComponent >>$ where the list contains all the remote actors that have received this future as part of an asynchronous method call.
3. The received future is stored by the actor in the *remoteUncompletedFutures* map.
4. When the future is completed either by:
   - a local actor.
   - a remote actor as explained in the protocol before.
   - a remote actor explained in the next step.

   the list of machines that require the future is retrieved and the entry in the map is removed.
5. The actor sends a pair $< futureID, result >$ to all the machines in the list that require it.
6. When a machine receives such a pair it completes the future identified by the *futureID* with the result and possibly runs steps 3 and 4 itself if it propagated this future as well.

19.3.3 Actor Execution Context

Actors run in a parallel and distributed environment through simple messages that are presented in Fig. 19.3. In addition to the usual object-oriented implementation, an actor exposes a method *send* which allows it to receive asynchronous method calls form other actors and this provides parallel execution between the actors. The class simply creates a lambda expression that takes the form of a Java Runnable or Callable and subsequently a wrapper future which may be used for synchronization purposes. In our previous version of the API, each actor had an execution lock that limited it to one method running at a time.

For a single machine, there was a single thread, called a *Sweeper*, available across all actors, that continuously checked all "unlocked" actors and submitted the head of their queue to the executor service. Actor execution is now demand-driven as shown in Fig. 19.4, with a single thread that is spawned into the state *ready* once the first message is received in the actors queue, moves to state *execute* and runs all messages in the queue and goes into state *stop* once the queue becomes empty, restarting once

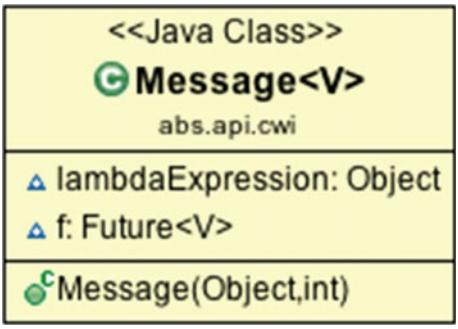

Fig. 19.3 Message encapsulation

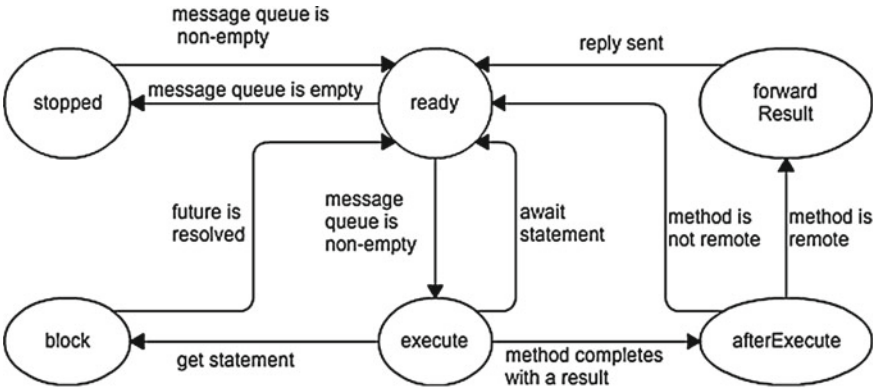

Fig. 19.4 Actor state diagram

another message is inserted in the queue. This makes actors completely independent from each other unless they explicitly call the synchronization mechanisms *get* and *await*.

19.3.4 Synchronization and Cooperative Scheduling

A key feature of the *Sweeper* thread was that it allowed efficient scheduling of tasks within an actor. It prevented redundant thread creation by having suspended tasks of an actor given priority once they were released to compete for the actor's lock. With the *Sweeper* thread deleted from the model of the API, we introduce a new mechanism to support cooperative scheduling. First of all when a *get* statement is called on a future, the actor moves to the state *block* until the future is resolved. If an *await* statement is encountered, the actor invokes another exposed method *await* which receives a boolean condition or a future to suspend on and a continuation in

the form of a lambda expression. The actor will then store a mapping of the continuation and the condition or future in a separate map as either *futureContinuations* or *conditionContinuations* specific to each actor and moves to state *ready*. The main executor introduced in the library is now responsible when, a thread completes, to run the *afterExecute* method which verifies if the method is remote in which case it has to *forward the result* to the socket from which it came to avoid a busy-waiting thread that may do this work. If the method invocation is from a local actor, the after execute method has to search each actor's continuation maps to identify the continuations that may have been resolved by this method (either an existing boolean condition or the actual future that has been resolved).

19.4 Description of the Preferential Attachment Algorithm

In this paper, we represent social networks through the notion of a graph where nodes are the members of the network and edges are the connections between them. The notion of Preferential Attachment (PA) is a specific model of adding a new member preferentially to a social network. We consider the above-mentioned preference to be the degree of the nodes, that is, roughly speaking, the more the degree of a node in the existing graph, the higher probability that it makes a connection with the new node. In this model, an existing graph of n nodes has a discrete probability distribution for the nodes with the probabilities $P_1, P_2, \ldots, P_n$ where $\sum_{i=1}^{n} P_i = 1$ and

$$P_i = \frac{deg(i)}{\sum_{j=1}^{n} deg(j)}$$

where $deg(i)$ returns the degree of the node i. One of the existing nodes is then randomly selected based on the above distribution to make connection with the new node. The Barabasi–Albert model [4], which is designed to generate scale-free networks using the preferential attachment mechanism, is one of the most commonly used models to produce artificial networks, because of its explanatory power, conceptual simplicity, and interesting mathematical properties [13]. The procedure to generate a PA-based graph with n nodes starts with a given initial clique with $m0$ nodes (a small complete graph). The remaining nodes are then added to the graph so that each new node makes m distinct connections with the existing graph ($1 \leq m \leq m0$) based on the distribution. The nodes are added sequentially (i.e., addition of the next node starts after terminating the addition of the current node) since, as shown in the above definitions, adding each new node influences the whole distribution.

Adopted from *Copy Model*, [8], we employ the array data structure to represent the graph. As depicted in Fig. 19.5a, from left to right each pair of array slots represents an edge of the array. In order to set up an array which represents the graph with the above-mentioned parameters, the array size is

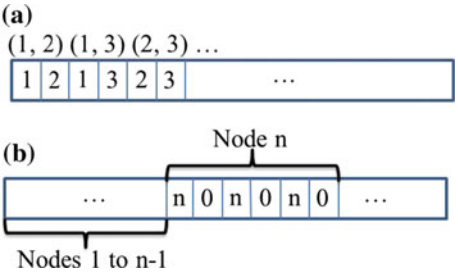

Fig. 19.5 The array representing the graph

$$S = init + 2m * (n - m0)$$

where *init* is the size of initial graph which can be a complete graph, $init = m0 * (m0 - 1)$. Figure 19.5b shows an abstraction of the array where the node n will be attached to the existing graph and $m = 3$. The array can be optimized in terms of memory since one slot of each pair for all the edges is calculable (e.g., n in Fig. 19.5b). However we ignore this optimization in this section. The next step is to resolve the unresolved slots for the node n (depicted by 0) according to the probability distribution of the existing nodes (i.e., $P_1, \ldots, P_{n-1}$). We simply use a uniform distribution over all the slots placed previous to the slots regarding node n since the number of occurrences of each node equals to its degree. Note that the values for the three unresolved slots must be distinct, which is simply checked via a function.

The sequential solution is fairly straightforward. Given the array with the initial graph at the beginning slots, according to above properties, the sequential solution is achieved via adding the nodes sequentially to the array.

However, the solution is more challenging in a parallel or distributed setting. To this aim, first of all the nodes (and the corresponding array) should be partitioned so that each partition is resolved by an actor. In the array, each node is represented by a sequence of slots where the first slot of each pair is the node's id (e.g., the slots regarding node n in Fig. 19.5b). If we consider all the partitioned arrays to be one virtual global array (like what we expect in the sequential approach) then the direction of dependencies and computations is depicted in Fig. 19.6. The arrow x in this figure shows a special kind of dependency, unresolved dependency, which

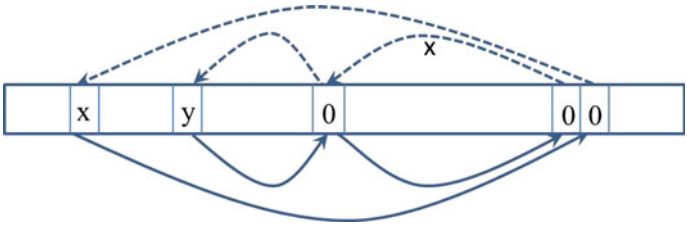

Fig. 19.6 The direction of dependencies (*right to left*) and computations (*left to right*)

shows the slot whose resolution is dependent on yet another unresolved slot, *target* slot. It is not difficult to see that unresolved dependencies only appear in the parallel solution. In order to remain consistent with the original PA model, the distributed approach keeps unresolved dependencies and uses the value of the target slot when it is resolved. How to keep the dependencies and use their target results after resolution is a low-level challenge. Figure 19.7 shows two different strategies to deal with this challenge. The first approach (Fig. 19.7a) is already examined in [1]. The second one (Fig. 19.7b) is adopted from [2], which is for multicore architecture, and tailored to fit the distributed setting. In the former case, the actor **b** places the request explicitly in a data structure and replies to it when the corresponding slot is resolved by the Actor. On the other side, actor **a** needs to keep track of the number of required responses corresponding to the requests. The latter however does not require such low level explicit synchronization management since it utilizes the notion of cooperative scheduling [5] via *await* on boolean conditions [2] to introduce a higher level of abstraction. Our implemented model follows the latter case.

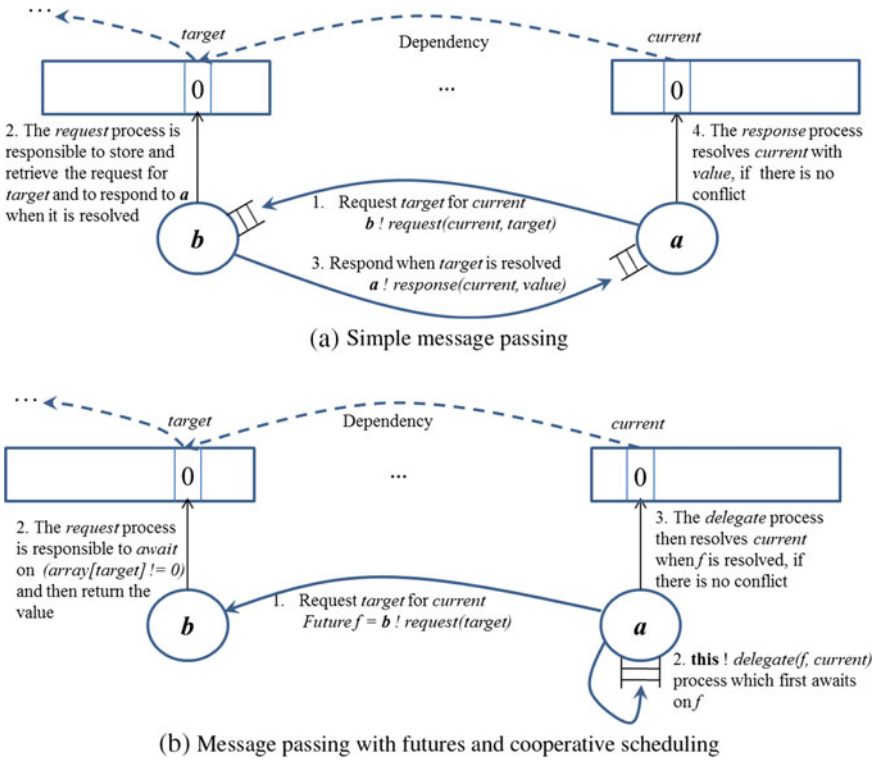

Fig. 19.7 The process of dealing with unresolved dependencies in an actor-based distributed setting

19.5 Implementation of the Algorithm Using the ABS-API Library

The implementation of the preferential attachment algorithm assumes a setting where the number of machines and actors is established a priori such that the application can assign predetermined global names to all the actors. In this manner all machines already have *RemoteActor* references to the actors they need to communicate with and corresponding communication streams setup as soon as all actors are instantiated and initialized. Figure 19.8 specifies our solution in a high-level pseudocode, which

```
 1: Each actor O executes the following in parallel
 2: run(...) : void
 3: for each Node i in the partition do
 4:     for  j = 2 to 2m do j = j + 2 step
 5:         target ← random[1..(i − 1) ∗ 2m]
 6:         current = (i − 1) ∗ 2m + j
 7:         x = whichActor(target)
 8:         actor[x]! request(target)
 9:         this! delegate( f , current)
10:
11:
12: request(target : Int) : Int void
13: localTarget = whichSlot(target)
14: await (arr[localTarget] ≠ 0)
15:                                          ▷ At this point the target is resolved
16: return arr[localTarget]
17:
18:
19: delegate( f : Future, current : Int) : void
20: await f?
21: value = f.get
22: localCurrent = whichSlot(current)
23: if duplicate(value, localCurrent) then
24:     target = random[1..current/(2m) ∗ 2m]
25:                                  ▷ Calculate the target for the current again
26:     x = whichActor(target)
27:     f = actor[x]! request(target)
28:     this. delegate( f , current)
29: else
30:     arr[localCurrent] = value                                    ▷ Resolved
31:
32:
33: duplicate(value : Int, localCurrent : Int) : Boolean
34: for each i in (indices of the node to which localCurrent belongs) do
35:     if arr[i] == value then
36:         return True
37: return False
```

Fig. 19.8 The sketch of the proposed approach

represents the scheme depicted in Fig. 19.7b. Each actor is responsible to resolve one partition of the virtual global array.

As shown in Fig. 19.5b, each node (as a new node) is associated with $2m$ slots of the array. Each actor starts processing its corresponding partition. The way array is partitioned has a considerable influence on the performance since it has a direct impact on the number of the unresolved dependencies (e.g., Consecutive and Round Robin Node Partitioning). In this section we abstract from the partitioning details. To this aim, we introduce two functions in the code. The function *whichSlot(i)* returns the local index corresponding to the virtual global index i, and the function *whichActor(i)* returns the actor index whose associated partition contains the local index corresponding to the virtual global index i.

The process *request* suspends on the boolean condition until it evaluates to *True*. The continuation is then queued and activated according to the actor's scheduling policy. The process *delegate* is also suspended until the future f is resolved. $f?$ returns a boolean value which represents whether the future is resolved or otherwise. Therefore the await on the future suspends the process until the future is resolved. The exclamation and dot are for asynchronous and synchronous method calls respectively. Finally the method *duplicate* checks whether the obtained value will cause a conflict, that is, a node makes two connections to the same target.

19.6 Conclusion

In this paper we presented a library to generate executable code in Java for an actor-based modeling language with very fine-grained scheduling heuristics formal analysis and verification tools. In this implementation we added support for distributed actors, future propagation and significantly reduced the number of threads created and alive throughout the application's lifetime ensuring efficient memory consumption and performance. We offered an enhanced API for distributed communication and explicit control of synchronization. Our focus was on the abstract behavioral specification language which represents an excellent solution for modeling cloud applications and this implementation allows the language to be extended with cooperative scheduling capabilities and powerful scheduling algorithms. We presented the details of how our new solution uses the latest Java 8 concurrent library to map the ABS constructs that invoke the scheduler and also ensure transparency with respect to actor's locations. We motivated our contribution by outlining the implementation of a specific scenario for generating large-social network graphs which can be deployed in a distributed environment using this library. The next step to this scientific work is to integrate this implementation into the ABS compiler that is currently in use to translate ABS code into executable Java code and investigate how to provide syntactic sugaring for ABS asynchronous method invocations. This will allow the direct modeling of our case study using ABS and testing it against the state-of-the art implementation in MPI.

Acknowledgments Partly funded by the EU project FP7-610582 ENVISAGE: Engineering Virtualized Services (http://www.envisage-project.eu). Partly funded by the EU project FP7-612985 UPSCALE: From Inherent Concurrency to Massive Parallelism through Type-based Optimizations (http://www.upscale-project.eu).

References

1. Alam, M., Khan, M., Marathe, M.V.: Distributed-memory parallel algorithms for generating massive scale-free networks using preferential attachment model. Proceedings of the International Conference on High Performance Computing, p. 91. Storage and Analysis, ACM, Networking (2013)
2. Azadbakht, K., Bezirgiannis, N., De Boer, F.S., Aliakbary, S.: A high-level and scalable approach for generating scale-free graphs using active objects. In: Proceeding of the ACM/SIGAPP Symposium on Applied Computing, To appear (2016)
3. Bader, D.A., Madduri, K.: Parallel algorithms for evaluating centrality indices in real-world networks. In: International Conference on Parallel Processing, 2006. ICPP 2006, 539–550. IEEE (2006)
4. Barabási, A.L., Albert, R.: Emergence of scaling in random networks. Science **286**(5439), 509–512 (1999)
5. De Boer, F.S., Clarke, D., Johnsen, E.B.: A complete guide to the future. In: Programming Languages and Systems, pp. 316–330. Springer (2007)
6. Johnsen, E.B., Hähnle, R., Schäfer, J., Schlatte, R., Steffen, M.: Abs: A core language for abstract behavioral specification. In: Formal Methods for Components and Objects, pp. 142–164. Springer (2010)
7. Johnsen, E.B., Schlatte, R., Tarifa, S.L.T.: Modeling resource-aware virtualized applications for the cloud in real-time abs. In: Formal Methods and Software Engineering, pp. 71–86 Springer (2012)
8. Kumar, R., Raghavan, P., Rajagopalan, S., Sivakumar, D., Tomkins, A., Upfal, E.: Stochastic models for the web graph. In: Proceedings of the 41st Annual Symposium on Foundations of Computer Science, 2000, pp. 57–65. IEEE (2000)
9. Nobakht, B., de Boer, F.S.: Programming with actors in java 8. In: Leveraging Applications of Formal Methods, Verification and Validation. Specialized Techniques and Applications, pp. 37–53. Springer (2014)
10. Serbanescu, V., Azadbakht, K., Boer, F., Nagarajagowda, C., Nobakht, B.: A Design Pattern for Optimizations in Data Intensive Applications Using Abs and Java 8. Practice and Experience, Concurrency and Computation (2015)
11. Serbanescu, V., Nagarajagowda, C., Azadbakht, K., de Boer, F., Nobakht, B.: Towards type-based optimizations in distributed applications using abs and java 8. In: Adaptive Resource Management and Scheduling for Cloud Computing, pp. 103–112. Springer (2014)
12. Serbanescu, V.N., Pop, F., Cristea, V., Achim, O.M.: Web services allocation guided by reputation in distributed soa-based environments. In: 2012 11th International Symposium on Parallel and Distributed Computing (ISPDC), pp. 127–134. IEEE (2012)
13. Tonelli, R., Concas, G., Locci, M.: Three efficient algorithms for implementing the preferential attachment mechanism in yule-simon stochastic process. WSEAS Trans. Inf. Sci. Appl. **7**(2), 176–185 (2010)
14. Yoo, A., Henderson, K.: Parallel generation of massive scale-free graphs (2010). arXiv preprint arXiv:1003.3684