Implementation of the lightweight probabilistic broadcast algorithm using RePast Symphony

Michele Armellini
203828
michele.armellini@studenti.unitn.it

Marco Merlin
205263
marco.merlin@studenti.unitn.it

1. Introduction

In this report are presented the implementation of the lightweight probabilistic broadcast protocol[1] for message delivery in distributed systems and the comparison between the results obtained in a simulated environment and the results presented in the original paper describing the algorithm. This project is part of the Distributed Systems 2 course offered by the University of Trento. The installer is found at `https://drive.google.com/file/d/1PrbhJsccTElbgLRMwKmzEO1jmlZVkm1v/view?usp=sharing`

The lightweight probabilistic broadcast algorithm tries to achieve distributed message broadcast in a scalable manner. The algorithm is mainly based on concepts commonly found in generic probabilistic broadcast algorithms such as gossiping, message buffering and membership. The main improvement brought by *LPB* is a more lightweight and scalable approach to membership management and message buffering.

To simulate a distributed environment the Repast Simphony[2] agent-based simulation framework was used. This java based framework allows to simulate many different scenarios depending on the needs. The framework allows to visually model all the possible behaviours observable in a distributed environment and described in the original paper such as joining and leaving of members, sending of messages, connections between subscribers and crashes.

The report firstly discusses the algorithm itself as it is described in the original paper and how the various mechanisms were implemented and represented in repast with a focus on some critical decisions and changes that were made to the original algorithm to either improve it or to simply adapt it to our simulation environment, it then presents how our implementation behaves with different parameters and how the results obtained in the original paper differ from the results obtained in our simulation.

2. Implementation with repast

This section briefly describes the core mechanics of the *LPB* algorithm, following is a description of how the details of the algorithm were implemented and how the environment was graphically modelled in repast.

2.1. Algorithm description

The algorithm described in the paper proposes a lightweight solution for message broadcast in distributed systems. The algorithm is based on probabilistic broadcast approaches which use gossip messages between peer to peer sets of processes. The key idea behind these probabilistic algorithms is to decentralize the broadcast process by sacrificing reliability for scalability. In order to distribute messages the processes send gossip messages containing the messages they received recently from other participants, in turn the recepient of these gossips can send these messages to other processes and so on. Eventually, the probability of all processes to receive a message is very high while the probability of a process to not receive a certain message is very low.

The probabilistic part of these kind of gossip based algorithms lays in the fact that the set of known processes participating in the system is limited for each process and is called the *view* of a process. This allows the system to scale well even with a high number of processes since each process has to know only a few of them and not all. When a process wants to gossip it will only send a gossip to a randomly selected subset of its *view* called *fanout* or gossip subset. The *LPB* algorithm differs from the generic gossip algorithms mainly in two ways. The first it's the employment of a completely randomized and fixed view where similar gossiping algorithms may instead employ variable size hierarchical views. The fixed view size makes this algorithm even better at scaling compared to similar algorithms since the randomized view choice allows this algorithm to be completely decentralized so that no process depends on any other process. To further improve randomization and keep uniform views, each process attaches a list of known processes to each of

their gossips so that the receiving processes can update their lists of known processes with elements from the newly received list. The second difference with regular gossiping algorithm is the buffering of data such as subscribers, unsubscribers, messages to send and messages received. All this data is periodically purged so that each process maintains a constant memory usage. Two purging methods are described in the original paper, one randomized and one age-based, both of these methods are described in the following subsections. Two other important mechanisms handled by the algorithm are members subscriptions and unsubscriptions. A joining member should know at least one already subscribed member. The new member should send a subscription message to its known process which in turn has a chance to spread this subscription information through a subscribers list sent out with each gossip. This way the new member will be gradually introduced to the other processes which will eventually start to send gossips to it. Similarly, unsubscribing members will send an unsubscribing message to processes in their view before exiting the system, unsubscriptions are spread through gossips as well gradually removing the unsubscribed process from all views and buffers of each node.

Another algorithm concern regards the recovery of messages. When a process receives a gossip it delivers the contained messages and saves them to a buffer so that it can verify whether a message has already been delivered. When a process sends a gossip it attaches a fixed number of messages, these messages are then wiped meaning that a process only sends the same message once. This is because the messages are supposed to be expensive to send. Each message, however, contains a unique id across all the environment. A fixed number of the last received message ids are also attached to each gossip. When a process receives a gossip it also checks whether the message ids advertised in the gossip are present in its delivered messages, if this is not the case the process recognizes that it is missing a message. A process with missing messages will try to contact different processes to try and recover the lost message. The heuristics used to choose which process to ask the missing message from are described in the following subsections.

2.2. Graphic Environment

The repast symphony agent-based simulation framework allows to visually simulate all kinds of interactions between different agents. To simulate a distributed environment a continuous space was used. When a process joins the system it is add to the space as a blue circle and assigned some random coordinates. The gossips themselves are represented as red squares which are spawned on the process which generated them and move with a fixed speed towards their target process. Recovery messages are represented as green squares with the same visual behavior as gossip messages. Finally, gray arrows are used to represent the connections between processes and how they change during the simulation. These arrows are updated each time a process receives a gossip and updates its view. Finally crashed or unsubscribed processes have a label underneath them for recognition and unsubscribed processes are also turned into red stars to be even more distinguishable.

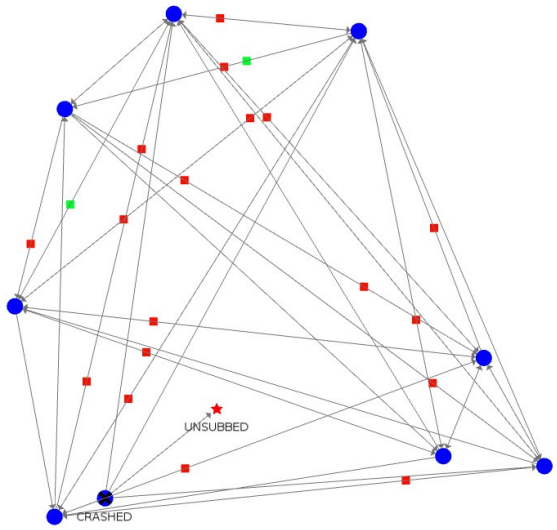

Figure 1. Screenshot from the simulated environment with the Repast Symphony framework

2.3. Algorithm implementation

In this subsection are presented several details regarding the *LPB* algorithm implementation. The way a single process is represented is a static entity which receives and sends gossips. It should be noted that the messages which should be delivered to the application are called events in our implementation and that is how they will be referred from now on. Each process contains its view as a list of references to other processes. The history of delivered events together with the buffer of events and event ids to send with the next gossip are also information stored in each process. Finally each process also has a subscribers and unsubscribers buffer to keep track of members joining and leaving and maintaining uniform views among processes. Similarly the gossip messages contain an event, event ids, subs and unsubs buffers so that all these information can be propagated in a single message.

The first important aspect of the algorithm is the gossiping itself which happens synchronously on each process at fixed time intervals called rounds. The first step for gossiping is choosing which processes should receive the gossips by selecting a random fixed size *fanout* subset from the view. The gossip itself is composed of four data structures: the events, the event ids, subscribers and unsubscribers.

Each time a process sends a gossip it copies its buffers into the message. Note that the events are only gossiped once and then purged, this means that a process can only gossip processes which it just received, but it is still possible that old event ids will be gossiped around more than once. The gossip themselves are separate actors with respect to the processes. When gossips are created they have a reference to both the target receiver process and the sender process. The receiver is needed for the gossip actor to understand which direction he needs to travel to, while the sender is needed by the receiver to know which process sent the gossip. As soon as a gossip is created, it will start moving towards the receiver process at a fixed speed, it should be noted that the propagation speed is always smaller than the gossiping intervals meaning that every gossip spawned in round *n*, will be received by its target before round *n+1*. The way this constraint was implemented is by a linear dependency between the size of the space and the speed of the gossips; the bigger the space, the faster the gossips. Once a gossip enters the proximity of its target the receiving process recognizes that the gossip is addressed to him because of the internal target reference stored in the gossip itself. At this point the receiving process will physically remove the gossip from the space and process it.

Another bug aspect of the algorithm is the gossip reception handling. The original paper subdivides this process into three steps. The first step consists into handling the unsubscriptions carried by the gossip. This involves adding all the unsubscribing processes to the unsubs buffer of the process and removing them from both the process view and subs buffers. The second step focuses on the handling of subscriptions carried by the gossip. This step consists into adding the new subscriptions to the receiving process subs list so that it will then be able to further advertise the new members, and picking random elements to add to the view so that uniform views among all processes can be achieved. Finally the third step consists into handling the events and event ids contained in the gossip. This step firstly takes all the new events contained in the gossip and delivers them to the application layer, it then proceeds to check if all the event ids contained in the gossip have already been received by checking the list of already delivered events. If the gossip contains an event id which has not been delivered yet, the receiving process will insert it into a retrieval list. Finally the events and the event ids contained in the gossip are saved to the events and event ids buffers so that they can be spread to other processes in future gossips. The way the paper handles gossip reception however, is not optimal. The unsubscriptions carried by a gossip are handled by removing them from both the view and the subs buffer and subsequently adding them to the unsubscribers buffer of the receiving process. As an example of how this behaviour could be bad suppose that process *p1* receives a gossip from process *p2* containing process *p4* as an unsubscriber. Following the paper's algorithm *p1* removes *p4* from both its view and subs buffers and then adds it to its unsubscribers buffer. Suppose now that process *p1* receives a gossip from process *p3* which contains *p4* as a subscriber. At this point process *p1* will start advertising process *p4* in its gossips as if it never received its unsubscription. This scenario can happen very easily since unsubscriptions take some time to propagate. To avoid this problem we choose to modify the gossip reception handling procedure described in the paper by modifying how unsubscribers are handled. Specifically we choose to move the unsubscription handling process after the subscription handling step, this way unsubscriptions can act on the updated view and subs buffers. Moreover unsubscriptions contained in the gossip are simply add to the unsubs buffer of the process and finally we remove each element contained in the updated unsubs buffer from both the view and the subs. By doing this each time a gossip is received the process is going to remove its unsubs from both the updated view and subs buffer. As shown in Section 3, our method greatly decreases the average time taken by a process to be completely forgotten by other processes in the system. It should be noted that this problem can still occur for crashed processes since they may have outdated information about the processes in the system.

Another very important aspect are the purging heuristics for processes buffers. This is one of the key features of this algorithm since it is supposed to be scalable and still work efficiently. To do so, all the processes buffers are purged each time they are updated to maintain a predefined fixed buffer size. The first purging method described by the paper is completely randomized, meaning that random elements are removed from each buffer until the target size is reached. While extremely inefficient, this method is very simple and allows the algorithm to work. The second method described in the paper uses age information to purge older data first. This kind of purging heuristic is applied to all process buffers except the event id buffer. The age is an integer associated with a buffer element, each time the purge procedure is called on a buffer, the elements with the highest ages are removed first. The age is propagated together with its corresponding buffer element, this way receiving processes are able to correctly estimate how old a certain received element is. Both events and unsubscriptions ages are incremented by a sending process each time they are propagated in a gossip round. When a process receives a gossip it inserts new events and unsubs, together with their ages, in its buffers and updates the ages for already received elements. For subscribers and view buffers, however, the age is incremented by the receiving process when a received gossip contains elements which are already present in either the subs or view buffers. For all these matching elements the ages are incremented by one and it

should be noted that the ages for the two buffers are independent from each other, meaning that two same elements in the two buffers may have different ages. The reason for the difference in the update methodology is that events and unsubs are information which should be shared among all the system, this is because both an event and an unsub should only be received once per process and then forgotten. Subs and view buffers, on the other hand, are used to continuously randomize the connections among the members of the system and for this reason their age should be local to each process. When a new event or unsub is created their ages are set to zero and will be incremented to one as soon as they are propagated for the first time. It should also be noted that the original paper does not specify how to handle age of retrieved events. We thought it would be reasonable to set retrieved event ages to one since they may be still relevant to other processes and should have a chance to be spread. Originally the paper specified a different behaviour for unsubs purging. Specifically, following the paper, unsubs should have a time based purge, meaning that regardless of the unsub buffer size, an unsubscriber should be deleted after a fixed amount of time has passed since it was originally received. The problem with this method is that crashed processes will have no way of knowing about new unsubscribers if they were down for more than the time required to purge all unsub information from the system. With our implementation however the unsubs will only be purged if the unsubs buffer is full and a more recent unsubscriber appears, allowing unsubscribers information to last longer in the system.

The last big aspect of the algorithm is the event retrieval mechanism. As described previously, a process will store any event id, which has not been already delivered, in a retrieval list. Before sending a retrieval message for a missing event, a process will wait K rounds starting from the round in which the missing event id was received. At the end of each gossip round, each process will send out a retrieval message for each missing event which has not yet been received for K rounds. Initially, the target of these retrieval messages will be the process from which the missing event id was initially received. If after T rounds the missing event still hasn't been delivered, meaning that the target process has either crashed or unsubscribed, the process will start asking for the missing event to the node which initially generated it. If the origin of the event has already unsubscribed, as a last resort, the process will start asking for the missing event to randomly selected processes. Note that the original paper suggested a different approach to message retrieval. Specifically after the first attempt to retrieve the event from the sender, the process should keep asking a random process and the originator of the event if the first were not to answer. Our approach aims at quickly resolving message retrieval without resorting to asking random processes unless it is really necessary, meaning that the event originator has unsubscribed. This allows to contact the two processes which have the greater chance to answer back with the missing event, meaning that the number of retrieval messages sent before an event is retrieved is minimized. With our approach there may be some concerns about storms of retrieval messages targeting the originator of a specific event, this however is not really a problem since statistically most message retrievals are resolved by asking to the sender of the missing event id without resorting to the event originator process. It should be noted that for this whole retrieval process to work, information such as originator process should be stored in the event ids while the sender process should be stored by a process which wants to retrieve a missing event id it just received.

Now that we described how the main parts of the algorithm are implemented, it is important to also discuss how the environment management is handled. For this task a centralized approach was chosen. In our implementation a manager class is responsible for actions such as simulation setup, gossip synchronization, joining or leaving of processes, crashes and recoveries, event generation and central authority. The need of a central management actor most certainly defeats the purpose of a distributed protocol, however it should be noted that many of the tasks that this entity performs are mandated by the simulation we are proposing and would not be needed in a real scenario. Actions such as subscribing, unsubscribing, crashes and recovers are dictated by the single processes, while gossip synchronization is ideal but not really mandatory.

The first task of the central authority is to initialize the system by generating the processes and setting up their buffers in a randomized manner to simulate an already started system. The different events which could happen in the environment happen at parametrized intervals and are actuated by the central authority in a iterative way. It firstly applies an unsubscription routine where a random active process is chosen and unsubscribed from the system. The process will be marked as unsubscribed and will send out a last gossip containing its own unsubscription. Note that at this point the process will be visually replaced with another actor which serves three purposes. It allows to distinguish unsubscribed processes. It allows to visually see other processes gradually removing the unsubscribing process from their views, as the arrows pointing to the unsubscribed process slowly disappear. It then allows gossips and retrieval messages to be physically removed from the space as there still is an entity which can receive and remove them. After the unsubscription routine is over, the authority applies a gossip routine. This routine allows to synchronize gossips among all the environment by calling the gossip function described above on each subscribed process at once. A counter keeping track of the current round is also increased

so that the processes can use it for tasks such as event retrieval and age computation. Right after the gossiping routine, a cleaning routine is applied where all the unsubscribed nodes are checked and physically removed if no other process or gossip contains any kind of reference to them. The authority will then proceed to generate a fixed parametrized number of random events on randomly selected nodes, this is how events are generated in our environment to simulate the application layer. The authority will then crash a randomly selected active node. The selected process will mark itself as crashed and will not be able to process gossips, retrieval messages and will be unable to perform any action. Furthermore a crashed process cannot be selected for unsubscription. The reason for this is that a crashed process would not be able to inform other processes that it is leaving the system with its own unsubscription. Unsubscriptions however are not really needed as their only purpose is to remove processes from the system at a faster rate. After crashing a random process the authority will proceed to recover a crashed process marking it as active again. Note that crash and recovery actions are interleaved so that a crashed process will keep that state for some rounds before recovering. Last but not least the authority adds a new process to the environment. The newly introduced process will have a randomized position in the space and will start to gossip as soon as it is introduced in the environment. The subscription and unsubscription processes described in the paper require an ad-hoc type of message to perform any of these two actions. In our implementation we choose to just use regular gossips as we do not see the advantage in using two specific messages to perform these to actions when the same can be done with the subs and unsubs buffers contained in every gossip.

3. Implementation Analysis

In this section this implementation is compared to the original one, highlighting the differences and making some theories of what causes different behaviours. Mainly three different aspects will be evaluated: the performance of the algorithm with different fanout values, the speed at which a new node is fully integrated in the network and the time a node takes from when it decides to unsubscribe from the network to when no other node references it in any way.

3.1. Fanout value analysis

In the original paper[1], the algorithm implemented is analyzed with different fanout values in order to understand what value seems to be optimal and if there are significant differences in the speed at which one message is propagated to everyone. In Figure 2, the original analysis can be seen. These results are obtained by analyzing the relation between different fanout values F and the number of rounds it takes for a single event to be broadcasted to the whole system composed by n = 125 processes. As discussed in the original paper, the figure shows that a higher fanout number decreases the number of rounds needed to propagate the event to all processes. This however has an upper limit, as if the value is too high there will be redundant messages through all the network, not contributing to the spreading of the event.

In order to verify the results obtained by this implementation, the values used in the graph are obtained experimentally, by using the mean over several tries in order to have a graph consistent with real results and resistant to deviations from it due to randomness.

Something worth of note is the fact that the plot of the original paper discusses expected values which, in our experience do not seem they can be considered as *average* values, since randomness is involved and in most cases the actual number of rounds taken is far higher than the value expected. For example the only node missing the broadcasted event may crash and take several rounds to recover, increasing the total number of round substantially. Data in Figure 3 was taken in a network with 125 nodes, each with a view size of 15 nodes. It has been found that the view size does indeed impact, even if in a little amount, the number of rounds taken to spread the event to the whole network, while the original paper says the opposite.

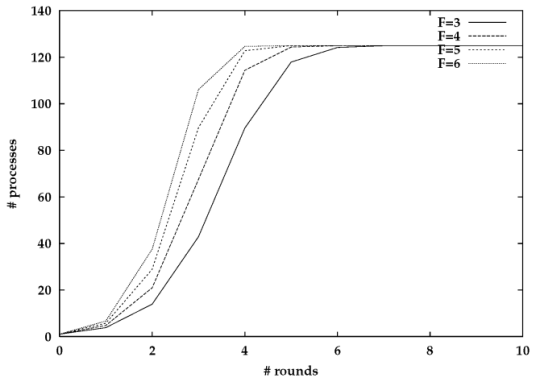

Figure 2. Expected number of infected processes for a given round with different fanout values in the original paper.

3.2. Node integration time

A different analysis on the algorithm is which evaluates the time it takes for a new process entering the network to have enough references to him and to reference enough nodes to be considered fully integrated. As with the previous plot, this one was made with a total of 125 nodes and with a view size of 15. The fanout value used is 3 and the maximum number of subscribing nodes and unsubscribing ones that a node can keep is 2. In the graph in Figure 4, on the x axis the round count from when the process decides to subscribe is shown, while on the y axis there is the node

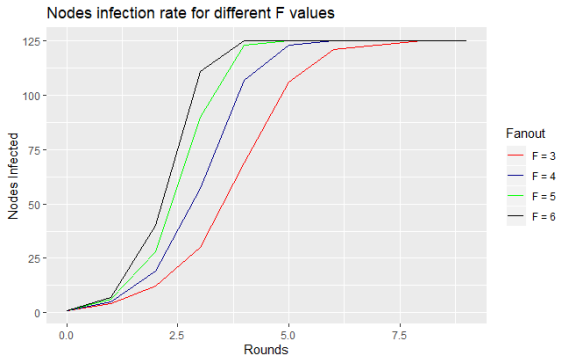

Figure 3. Number of infected processes for a given round with different fanout values in the implementation discussed in this document.

count. The blue line shows the total of processes the new node is referencing in its view, while the red line shows the total number of nodes that have the new subscribing node in their view. For both lines, the median value over the data collected from more than twenty different subscribing nodes is shown, in order to have data affected the least possible from randomness. From both the lines it can be clearly seen that the nodes take a while to successfully insert themselves in the network. This is of course mainly because of the small subscribing set size of each node. It has to be noted that randomness of the selection of processes to send data to plays a huge role in the real performance of the algorithm.

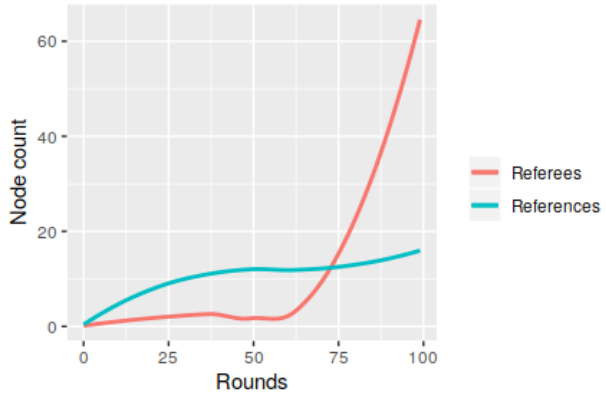

Figure 4. Progress in the integration of a node in the network after subscription

3.3. Node unsubscription time

The last analysis made on this algorithm consists in the comparison between the unsubscription described in the original paper and the different implementation described in Section 2.3, specifically the removal of the unsubs buffer elements from both the view and the subs buffers every time a gossip is received.

In Figure 5 the performance of the original implementation can be seen. It is easy to see that a node can take a lot of time to completely unsubscribe by making other nodes lose all references to it. This is mainly because after a lot of unsubscription, when the unsub set is filled, it is a matter of randomly choosing the values that can stay and those who are spread around and, if a node is ”unlucky”, it may take a really long time to be unreferenced.

In Figure 6 the performance of the algorithm described in this document is shown. As can be clearly seen, the average time a process takes to be fully unreferenced is notably lower than the one in Figure 5. Furthermore, while the original implementation sometimes has some processes taking more than 50 rounds to completely unsubscribe – and sometimes that being just because they are removed from the view by random chance and not because of an unsubscribe message–, the improved implementation has no single process taking more than nine rounds over a huge number of unsubscriptions.

This proves that, while the implementation described originally is working reliably, it has a lot of room for improvements.

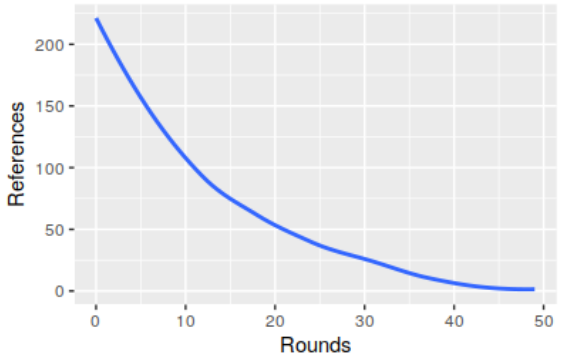

Figure 5. Progressive reduction of the references to a process that is unsubscribing with the algorithm used in the original paper

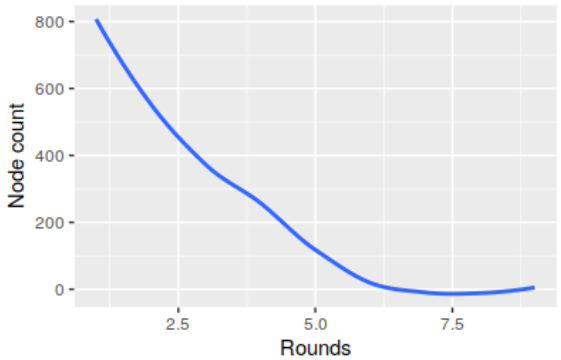

Figure 6. Progressive reduction of the references to a process that is unsubscribing with the algorithm used with the optimized algorithm described here

4. Conclusions

To conclude in this paper was presented the implementation of the lightweight probabilistic broadcast algorithm described in [1] and an analysis of the results obtained in a simulated distributed environment. In Section 2 were discussed the general mechanics of the algorithm and how they were implemented in the Repast framework with a focus on explaining the changes we thought necessary in order to perform the simulation or to optimize the overall performance of the algorithm. The results presented in Section 3 show results almost identical to the ones obtained in the paper. Furthermore, we presented some graphs showing the results for some of the changes made to improve the original algorithm.

References

[1] P. T. Eugster, R. Guerraoui, S. B. Handurukande, P. Kouznetsov, and A.-M. Kermarrec. Lightweight probabilistic broadcast. *ACM Trans. Comput. Syst.*, 21(4):341–374, Nov. 2003.

[2] M. J. North, N. T. Collier, J. Ozik, E. R. Tatara, C. M. Macal, M. Bragen, and P. Sydelko. Complex adaptive systems modeling with repast simphony. *Complex Adaptive Systems Modeling*, 1(1):3, Mar 2013.