Repast HPC Social Media Platform Emulator (RHPC SMPLE)

User Guide

November 22, 2021

Contents

Chapter 1

Introduction

Repast HPC Social Media Platform Emulator (RHPC SMPLE) is a toolkit for building agent-based models of behavior and information spread on social media platforms. The toolkit allows creation of a simulated social media platform with users who can engage with that platform, share content, and interact. The models created are agent-based models in the sense that the users who are interacting with the platform are agents and the platform itself is an agent. As agents, platforms can make decisions about what information they share among users; users as agents can make decisions about how they use the social media platform. The simulated platform can provide functionality representative of real-world social media platforms (e.g., posting, liking, rating, following, etc.).

RHPC SMPLE is constructed from Repast HPC. Repast HPC is a toolkit for building large-scale agent-based models that can be parallelized for execution in high performance parallel computing environments. When an agent-based simulation is parallelized, the parallelism results in a challenge of keeping simulation states consistent across processes while allowing parallel execution of portions of the simulation on each process. Repast HPC handles the inter-process communication and synchronization issues in flexible and customizable ways while allowing the model developer to avoid writing low-level parallelization code. RHPC SMPLE is written in C++ to be usable on Top 500 HPC systems, potentially allowing social media simulations at real-world scales, i.e., millions or billions of agents.

1.1 How to Use this Manual

This manual provides a comprehensive guide to building social media emulators in RHPC_SMPLE and to deploying and benchmarking these emulators. The guide consists of the following chapters:

Chapter 2, Installation, provides technical details for installing RHPC_SMPLE, including requirements and dependencies.

Chapter 3, Overview of Functionality, provides a detailed description of the functionality of the RHPC_SMPLE toolkit.

Chapter 4, Code Structure, gives a tour of the source code for the toolkit and how it is organized.

Chapter 5, Create a New Platform, gives step-by-step instructions for creating a new abstract social media platform in the RHPC_SMPLE toolkit.

Chapter 6, Extend a Platform to a Demo, gives instructions for creating a specific implementation of a social media platform in the RHPC_SMPLE toolkit.

Chapter **??**, Create User Agents, describes how to create agents that use the implemented social media platform.

Chapter 7, Create Scenarios, gives instructions for how to create scenarios that encapsulate a specific use case for a model of one or more social media platforms, users, and exogenous events to which users and platforms respond.

Chapter 9, Output, discusses how to create structures that output to files.

Chapter 10, Launching a Simulation, describes how to launch the simulation (including parallelized versions) and how to specify simulation options from the command line.

1.2 Acknowledgements

Chapter 2

Installation

Use of RHPC_SMPLE requires the installation of the Repast HPC toolkit. Repast HPC carries three main requirements:

> **The Boost C++ Library**: the Boost[1] library is a set of C++ extensions on which Repast HPC relies heavily; RHPC_SMPLE is tested with versions up through 1.76.
>
> **NetCDF**: NetCDF[2] is an extension to C++ that allows data read/write in a special dense format widely used in high-performance computing.
>
> **curl**: The curl library[3] is a library that permits file transfers using multiple protocols.

Additionally, Repast HPC must be installed in a system that has an implementation of MPI (Message Passing Interface). MPI is a system that allows launching programs across multiple processes and manages the communication that occurs among these processes. Implementing MPI is a standard parallel processing operation, so any HPC system will likely have this installed; consult your system administrator. It may be possible to install MPI on a laptop, desktop, or AWS instance. Common implementations of MPI are OpenMPI[4] and MPICH[5].

Repast HPC can be installed by compiling against, and linking to, the Boost and NetCDF libraries and using your system's appropriate MPI compiler.

Once this is done, RHPC_SMPLE can be installed by compiling against, and linking to, the Boost, NetCDF, and Repast libraries using the makefile included. To use this makefile,

[1] www.boost.org
[2] https://www.unidata.ucar.edu/software/netcdf/
[3] https://curl.se/libcurl/
[4] www.open-mpi.org
[5] www.mpich.org

modify the file *rhpc_smple.env* and update the locations of Boost, NetCDF, and Repact HPC on your system. When this is built, it will create a new library, rhpc_smple.lib.

Once RHPC_SMPLE is installed, you can compile using the header files in the *include* directory, and build linking to the rhpc_smple.lib library.

Chapter 3

Overview of Functionality

At a high level, the RHPC_SMPLE toolkit allows creation of simulations of user behavior on social media platforms. Within this high level are a number of specific details that are summarized below.

3.1 Overview

The RHPC_SMPLE toolkit is designed to create social media simulations that are conceptually structured as a set of concentric containers:

> At the outermost level is the **model**. A model is a runnable object that can be called by a c++ mainfunction. It executes the overall schedule of the simulation from start to completion.
>
> Each model can contain a **scenario**. A scenario is a collection of social media platforms and any data that represent the real-world within which those platforms exist (e.g., world news events that will be discussed on the platforms). Scenarios can be nested so that a single scenario can contain multiple other scenarios.
>
> A scenario contains one or more social media platforms.
>
> Platforms contain nodes in a network. Generally, these nodes are of two types:
>
> – **Content nodes**, which represent posts to the social media platform, and
>
> – **User nodes**, which represent user accounts on the social media platforms

3.2 The Model

The model is a container that wraps and guides the overall execution of the simulation. It handles the initialization of all the scenarios it contains (generally just one, but that one could potentially contain other nested scenarios; see below), the initialization of the simulation schedule, the overall management of time within the simulation, and any other required aspects of construction and tear-down.

Commonly, the model object is generic to a particular research project, and this can be written to contain multiple variant scenarios, all of which have time managed in the same way and all of which require similar schedules, initialization, and tear down, so that the scenario objects can be swapped in and out as needed.

3.3 Scenarios

Scenarios contain platform objects and representations of the real-world within which those platforms exist. The details of platform objects are discussed below. What is meant by the representation of the 'real-world' can vary widely from project to project, but some examples could include:

> A time series of prices in some commodity, which agents on the platforms could then discuss
>
> A set of news events that occur during the course of the simulation, which agents on the platform could respond to by increasing their activity levels
>
> A time series of outages (accidental or intentional) that prevent some agents from participating in the discussion for periods during the simulation
>
> A collection of specific pieces of information that would be available for discussion by the agents, and/or inserted into conversations on the platforms at specific times

In each of these cases, the scenario would be responsible for initializing the data structure that represents these items and for managing their statuses and changes during the operation of the simulation.

3.3.1 Nested Scenarios

A scenario can contain other scenarios. This employs several principles:

> The model object need only call methods like ‘initialize’ on the outermost parent scenario; parent scenarios pass these downward to their child scenarios, so that all of them are initialized and operated in synchrony.
>
> Information that is available to parent scenarios is also available to child scenarios.
>
> Scenarios can be nested on-the-fly. At run time, a scenario can be designed and implemented such that it may or may not contain child scenarios so that tests can be run with and without the child scenarios. Scenarios can be designed once and re-used in different combinations.

The specific procedures to implement these are discussed in the technical documentation given below.

3.4 Platforms

The core of the RHPC_SMPLE toolkit is the ability to construct a social media platforms. The Repast HPC paradigm for creating agent-based models relies on a core container object called a **context** which serves as the container for all kinds of agents[1]. With a context exist one or more ‘projections’, which contain the information about how agents are related to one another, either in a spatial relationship (e.g. a grid space) or connected in a network. In the RHPC_SMPLE toolkit, a platform is a wrapper around a context and contains one core projection. The default implementation must only define the nodes that exist in this network, and the characteristics of the network’s edges. Edges, for example, can contain ‘following’ relationships among users, or ‘liking’ relationships between users and content, or any other element that the platform needs to fulfill its set of functions.

These functions are constrained and specific, and they exist at two levels. At the generic level, platforms should be able to create new users[2], and they should be able to respond to **queries**, which are requests for information about content or users on the platform. At the specific level, the platform should provide an **Application Programming Interface** (API) that user account agents can use to perform actions on the platform. These actions will include creating new content, forwarding or sharing existing content, etc., and should reflect the actions available in the real-world simulation being modeled[3].

[1] This conceptualization is derived from the Repast Java implementation but has a critical di erence: in the Java version, contexts can be nested, so that one context can contain another; in Repast HPC this is not true, in part because contexts serve as the instrument for parallelization, a function that would be complicated by this nested structure.

[2] Platforms should also be able to create existing content nodes, especially if the operation is parallelized and the updates are to non-local nodes that are copied on the local process.

[3] The degree to which an implementation of a speci c platform's API re ects the real-world API of that platform is a design choice, and the implementation will not necessarily be fully equivalent to the real one.

The platform's agency—that is, its ability to be proactive and strategic in managing its relationships with its user accounts—comes from the fact that it is able to strategically respond to queries (by, for example, customizing the results it passes when a specific user account requests information) and that it can respond to requests to its API by making (or declining to make) changes to its network content. The latter can include tracking specific actions on specific content with information that can later be used to respond to queries in strategic ways. For example, if 100 accounts use the platform's API to indicate that they 'like' a specific piece of content, then when a user account issues a query to see new content on the platform, the platform can select this popular piece of content over other new elements as a way to drive engagement. Later, it can respond by promoting other content produced by the same user account that originally produced the popular content.

3.5 Social Media Network Nodes: Content and User Accounts

Platforms are fundamentally wrappers for RHPC's network projections, and hence network nodes are central to its operation. The graph used is a directed graph. Nodes can be of two kinds: **content nodes** and **user accounts**. The network need not be strictly bipartite: links are possible between any two kinds of nodes; the most common may be links between users and content, but links from content to users, users to users, and content to content are all possible. The network can be used to track these relationships, for example:

User A 'liked' Content Z

User A 'follows' User B

Content Z was created by User A

Content Z was a quote of Content Y

The development of network edges is a key aspect of creating a RHPC_SMPLE simulation. The edge data structure should be able to capture these kinds of relationships[4] and make them available for use by the API and for responding to queries.

[4] An advanced design can be employed that adds more than one network to a platform. This could be done for performance reasons, e.g. having a separate network that just stores content-to-content relationships may make searching for those relationships faster.

3.5.1 Social Media Content

Content nodes represent pieces of content that users have posted on the social media platform. In the abstract, these are placeholders for any component that expresses a relationship among users; their main function is to link user accounts to other user accounts. Without any semantic association, this is the only things that a content node can do.

However, content nodes, in the abstract sense, can be endowed with a *payload*, which represents real content: pieces of information that users can engage with or interact with, share and transmit, and respond to in specific ways. For example, a content node could be endowed with a true-or-false flag representing some issue that some agents are 'for' while other agents are 'against'. Content that is on one side of the issue may attract specific responses from some agents, and opposite responses from others; the platform can use this to strategically present agents with content of one kind or the other. Richer content (stance on multiple issues, combinations of specific pieces of information, or event full text) is possible; no real limitation is imposed by the RHPC_SMPLE toolkit.

3.5.2 Social Media User Accounts

User accounts reflect entities on the social media platform that can respond to information and take action. Actions are undertaken via the API. User accounts are typically agents, with strategies, goals, memories, and other attributes that guide their interactions with the social media platforms and with the other accounts that they encounter through those media. The key element that the RHPC_SMPLE toolkit is intended to simulate is how agents, pursuing these goals through their varying strategies, are able to use the social media platforms, and the resulting social structure and flow of information.

People and Bots

A specific technical issue in the above discussion is that user accounts on social media platforms are just that: accounts. Typically, these are considered to be 'agents', and thus we conceive of them as individuals with memory, action, intent, strategy, or whatever other elements of the social simulation we want to use. However, an account is not a person. In some simulations, it may be preferable to create representations of real-world people; each person, then, can possess and act upon one or more social media accounts[5]. Strategies and other aspects of behavior, including offline interactions, could in this way be implemented across social media platforms, and user accounts would merely be tools that these person objects would use. This design choice may be specific to the project being undertaken.

[5] It is also possible to have a single account that is shared among multiple people (such as di erent managers of a public-facing corporate account), but this is not discussed here.

Because it is often difficult to fully link user accounts represented in social media data with real-world individuals who hold multiple accounts, the paradigm in which an account on a platform is the locus of agency (strategy, memory, etc.) is the more common case, but the use of person objects that possess multiple accounts is easily accommodated as well.

Bots, fortunately, are a simpler case, as there is essentially no distinction between a bot and a person, save that the capabilities and strategies of a bot may be very different. For example, a 'person' might be implemented such that actions by that person are only undertaken during waking hours of the day, while a bot would have no similar restriction. Bots could in theory control multiple accounts, or they could be implemented as if they were equivalent to user accounts (with no separate 'person' class possessing the account), as in the above discussion.

viddo thest(anardo)-333Repasto cp ase.mThsea incldh:

jopowtddoeleSelecution

Sele

ﬄ

ﬄ la

Chapter 4

Code Structure

The RHPC_SMPLE toolkit distribution includes three major components, and this chapter provides a brief overview of the functionality of each:

Social Network Tool Kit: This is the core code that provides the fundamental functionality from which so474(Ne(funee-257(incvlatfh)1orom)-334emc)28ulatores p uilt.:

ﬄImpleamethisEptiosstm-l
tctiost uilta from the core coh

ﬄactilsidemiostractiostcoostfruncetheit
omponents bpovc

Other utilities (***Utilities***)

Weighted Selection

Weighted selection is a common operation in simulations. For example, assume that there are ten agents that have an instance variable called 'salary'. The goal is to simulate that agents with a higher salary may perform some action (e.g., buying a luxury item). The simulation to permitted to choose an agent randomly. If an even distribution is used for selection, every agent has an equal probability of being selected:

$$p_i = C \tag{4.1}$$

where the probability of agent i being selected is equal to C, a constant. However, suppose the selection is weighted based on salary, such that:

$$p_i = \frac{s_i}{\sum\limits_{a=1}^{n} s_a} \tag{4.2}$$

That is, the probability of agent i being selected is equal to that agent's salary divided by the sum of all agents' salaries. Assuming salaries of:

Agent	Salary
Agent 1	20,000
Agent 2	10,000
Agent 3	5,000
Agent 4	5,000
Agent 5	5,000
Agent 6	5,000
Agent 7	5,000
Agent 8	5,000
Agent 9	5,000
Agent 10	5,000

Table 4.1: Agents and Salaries

The relative probabilities of any agent being selected are easily calculated from the salary values: Agent 1 has twice the probability of being selected as Agent 2, and four times the probability of any of the other agents (3 through 10). In terms of absolute probability, Agent 1 has a probability of $\frac{2}{7}$; this is because the total salary is 70,000, and Agent 1 has

20,000 of this. The others' probabilities are $\frac{1}{7}$ for Agent 2, and half of this ($\frac{1}{14}$) for all other agents.

To achieve weighted selection in the RHPC_SMPLE code, the SimpleWeightedSelector class can be used. The usage would be in three steps:

1. Create the instance.
2. Add elements to the instance in a specific sequence. The elements are the ***values*** that should be used as weights. In the example above, this would be the salaries.
3. Request a selection; the value returned will be the ***position*** (zero-indexed) in the list of values.

Importantly, the SimpleWeightedSelector takes only ***integers*** as weights.

The ParameterizedWeightedSelector operates exactly as a SimpleWeightedSelector after the values are added but allows parameters to be added that adjust the values as they are being input. This permits changes of several different kinds:

> An exponent can be added, such that the values that are put in are raised to a power. In the above example, this would cause Agents 1 and 2 to be even more heavily weighted. If the exponent was 2, then the weighting for Agent 1 would be 400,000,000; for Agent 2 it would be 100,000,000; and for all the others it would be 25,000,000. This means that the relative weight for Agent 1 would be increased from merely 4 times that of the lowest-salary agents to a factor of 16.
>
> A scalar multiplier can be applied in the ParameterizedWeightedSelector. Doing so does ***not*** change the relative weightings. Use this for cases where the original weightings are less than one; applying the multiplier transforms these weightings to integers (but truncates digits when doing so, which may make small distinctions between the weightings invisible).
>
> The weighted selector allows the weightings to be adjusted after they have been initialized. One use of this is selection without replacement. If this is needed, then after the selection is made and the index of the selected element is returned, a call can be performed to ***clearat()***, which will re-set the weighting at that position to zero. Once the weight is zero, that element cannot be selected again.

Return Values

As a convenience, the toolkit provides a collection of constants that are used throughout the code as return values in the case of unsuccessful termination. Use the ReturnValues.h class as a reference to the meanings of these.

FileWriter

A generic class for writing output to files is provided here; see Chapter 9 for more on writing to files.

NOTE: This class is specific to the RHPC_SMPLE toolkit. It is not a part of the social media structure, and utilities are not related specifically to the simulation of social media.

Other Utilities

One final class of utilities contains small set of functions that are used in the code. One is a helper function to extract a domain from a full web address. The others relate to setting properties with default values. See Chapter 7 to learn more about setting properties.

4.1.2 Core Tools

The core tools of the RHPC_SMPLE toolkit are contained in the socialnetwork_toolkit directories. There are 13 header files with four corresponding source files are present. They are shown in Table 4.2:

Header	Source
Action.h	
BehaviorSelection.h	
EdgeInformation.h	
EventCounter.h	
Explanation.h	
Feed.h	
InfoID.h	InfoID.cpp
InfoIDSelection.h	
InfoStore.h	InfoStore.cpp
Payload.h	Payload.cpp
Scenario.h	Scenario.cpp
SocialNetwork_AbstractElement.h	
SocialNetwork_Platform.h	

Table 4.2: Core Toolkit Files

The file listed in table 4.2, and the classes they contain, fall into a few categories:

There are the *core elements* from which platforms and user agents are built:

{ SocialNetwork_AbstractElement

{ SocialNetwork_Platform

{ EdgeInformation

{ EventCounter

There are the *actions that these agents can take*, including content that these actions might convey:

{ Action

{ Payload

There are those that deal with units of information that agents may discuss (e.g., hashtags):

{ InfoID

{ InfoIDStore

{ InfoIDSelection

There are those that represent the collections of data shared from platforms to users (to which users respond), and any explanatory elements that get transmitted with these collections:

{ Feeds

{ Explanations

BehaviorSelection provides a generic structure by which user agents can choose behaviors

Scenarios are containers that include platforms and 'real-world' elements and events to which agents and platforms may respond

4.2 Platform Implementations

4.2.1 Twitter

Twitter was founded in 2006 and is now one of the most widely used social media platforms in the world, with more than 192 million active users[1] and over 500 million messages posted

[1] https://s22.q4cdn.com/826641620/ les/doc_ nancials/2020/q4/FINAL-Q4'20-TWTR-Shareholder-Letter.pdf

every day[2].

Twitter is a microblogging platform that allows users to send and receive short posts. The length of these posts, which are called "tweets", can be up to 140 characters long. In these tweets, users can include emojis, images, videos, and URLs. Original tweets can be "retweeted", meaning information in a tweet from one user is shared by another user. Thus, retweeting is a quick and easy means of sharing information across wide audiences. A tweet has the capacity to reach a wide number of people.

Another aspect of tweets are that they can be replied to and quoted. Replying and quoting aren ways of engaging with users by starting, or joining, conversations. Another means of engagement is the "like" function. By liking, users can show acknowledgement, appreciation, or support for a tweet.

On the real Twitter platform, users can "follow" other users. Following subscribes a user to another user's tweets. This means that a follower will be able to see tweets on their "home timeline". The home timeline shows all of the tweets from all followed users. From here, a user can like, reply, quote, or retweet. Another feature of the real Twitter platform is that it enables the followed and follower users to send direct messages to each other.

RHPC_SMPLE Implementation

In the RHPC_SMPLE tool kit, users can undertake four actions: tweet, reply, quote, and retweet. There is no *follow* functionality, nor is there anything akin to a *like*.

4.2.2 Telegram

Founded in 2013, Telegram is a social media and instant messaging service meant for mass communication, but with a concern for privacy[3]. As of January 2021, Telegram has over 500 million active users. In 2021,Telegram was one of the most downloaded apps in Brazil, Malaysia, Russia, the Netherlands, India, Germany, the United Kingdom, Australia, and the United States[4]. In 2019, it was used for Hong Kong protests and by Islamic State extremists[5].

One feature of Telegram is that it has secret chats. These are end-to-end encrypted conversations, meaning that the conversations are private and cannot be monitored. Telegram also allows file sharing that can be done through these secret, encrypted chats. Another

[2] https://www.dsayce.com/social-media/tweets-day/

[3] https://www.spglobal.com/marketintelligence/en/news-insights/latest-news-headlines/telegram-crosses-500-million-monthly-active-users-62094638

[4] https://backlinko.com/telegram-users

[5] https://www.businessofapps.com/data/telegram-statistics/

feature of Telegram is that it has private and public “channels”. These channels are groups where up to 200,000 users can communicate[6]. For private channels, users have to be invited and permitted to join.

The RHPC_SMPLE implementation of Telegram is essentially identical to Twitter, save that the actions available are only ’post’ and ’reply’.

4.2.3 Reddit

Founded in 2005, Reddit is a collection of forums for different topics. There are different communities for different topics, and each community is called a “subreddit”. It is the seventh most popular website in the United Sates[7] and the nineteenth most popular worldwide[8]. Reddit is used for sharing content such as news, images videos, and text. These content types are posted to subreddits. There is a searchbar for searching key terms to find posts and subreddits to find content that relates to a given key term. It is also possible to search for users. As of 2020, there were about 222 million active users[9], and there are more than 46.7 million searchers a day [10] due to the breadth of topics available on Reddit.

“Commenting” is a way of engaging with content and other users. Comments can be responses to an original post or to other comments. When responding to a post or comment, it is possible for a user to “quote” part of that comment for the purpose of specifying what is being responded to. Also, similar to the “like” function in Twitter, “voting” shows acknowledgement, appreciation, or support via the “upvote” function. This also increases visibility. However, there is also a “downvote” function which is typically used for when a post is not appropriate for a given subreddit, and this decreases visibility.

Subreddits tend to have their own rules for behavior that are enforced by moderators, or “mods”, who volunteer to maintain the content and user behavior in the sub.

The homepage shows trending news posts from different subreddits, popular posts from different subreddits, and trending posts from the subreddits to which a user subscribes. A user can subscribe to other users too via the “follow” function. News can be sorted based on interests and date while popular posts from popular and user-selected subreddits can be sorted by upvote-to-downvote ratio, newness, and other features.

In the RHPC_SMPLE tool kit’s Reddit, users can post and comment. While the real Reddit has more user options, posting and commenting have been deemed the essential actions for understanding user behavior regarding spread of information.

[6]https://www.businessofapps.com/data/telegram-statistics/
[7]https://www.alexa.com/topsites/countries
[8]https://www.alexa.com/topsites
[9]https://www.oberlo.com/blog/reddit-statistics
[10]https://foundationinc.co/lab/reddit-statistics/

4.2.4 Jamii Forums

Founded in 2006, Jamii is a collection of forums for engaging in free discussions. Based in Tanzania, Jamii is known for its citizen journalism and for being an alternative news source. While there are a variety of forums for different topics, the popular forums discuss political, economic, and social issues as well as current events. Jamii is the top most visited website for Swahili speakers around the world, and as such, it is most popular in East Africa[11]. As of August 2017, there were "more than 2.4 Million users, 28 Million mobile subscribers and up to 600,000 people using its online forum daily"[12]. More recently, on June 25, 2020 at 11:42 AM CT, there were 1,554,248 threads, 38,986,717 messages, and 575,354 users[13].

The homepage consists of lists of forums. From here, it is possible to create new posts, search forums for key terms or users, see new posts and recent activity of other users, and see lists of online users.

Like Reddit, "commenting" is a way of engaging with content and other users. Comments can be responses to an original post or to other comments. When responding to a post or comment, it is possible for a user to "quote" part of that comment for the purpose of specifying what is being responded to. Also, just as with the "like" function in Twitter, users can engage with other users and their content via the "like" function. By liking, users can show acknowledgement, appreciation, or support for a post or comment.

The RHPC_SMPLE implementation of Jamii currently allows only posts, which initiate a thread within a forum, and comments, which belong to that thread. Quoting, which would have the effect of enabling one post to reply directly to another, is not implemented.

NOTE: As a technical detail, the RHPC_SMPLE Jamii implementation differs from the Reddit implementation in one important way: As a way to ensure performance at very large scales, the Reddit implementation does not treat all content as 'first class' content nodes. Instead, all 'posts' are full nodes, but 'comments' are recorded only as partial records- essentially a notation that something happened, but not a full record of the event. These notes descend from posts in a tree structure (comments of comments), but comments cannot be directly selected as targets of actions. The performance enhancement that this offers is mainly speed in selecting targets, as well as some reduction in the memory footprint of the simulation.

[11] https://www.zoomtanzania.com/company/jamii-forums
[12] https://tanzania.mom-rsf.org/en/owners/companies/detail/company//jamii-media-co-limited-1/
[13] https://www.jamiiforums.com

4.2.5 YouTube

Founded in 2005, YouTube is a site for sharing videos comprised of original content. The most popular types of videos are tutorials, comedy, and music[14], but there are other popular genres too. Users can also livestream to other users in real time. YouTube is popular worldwide and has local versions for over 100 countries and has over 2 billion regular users. Everyday, users watch "over a billion hours of video and generate billions of views" [15]. Additionally, YouTube averages 100 hours of uploaded video per minute[16]. Videos often have closed-captions that can even be available in multiple languages. Aside from hosting videos, YouTube has features that render it a social media platform. Each "channel" on YouTube is a user's account. Users can subscribe to other users' channels to stay up to date on other users' content. There is also a "comment" function so that users can comment on videos and reply to other comments. [17]. Additionally, users can "like" videos and comments to show acknowledgement, appreciation, or support. Liking boosts the popularity, and therefore visibility, of comments and videos.

On the homepage are recommended videos based on subscriptions as well as those that the platform thinks the user would like. There is also a section for news videos.

4.2.6 GitHub

Founded in 2007, GitHub is a leading code-sharing site. While not a traditional social media site, it is nevertheless a place where people interact online: "Millions of developers and companies build, ship, and maintain their software on GitHub" with over sixty-five million developers and over three million organizations involved [18].

The major features of GitHub are that it enables collaboration and maintaining version control for coding projects. Collaboration involves four main processes:

Creation and usage of a 'repository' of any and all project parts. This can include data sets, folders, files, images, videos, etc.

Creation and management of a 'branch'. There is always a 'main' branch. Branches are used to experiment with changes before 'committing' them to the main branch.

[14] https://www.thinkwithgoogle.com/consumer-insights/consumer-trends/top-content-categories-youtube/

[15] https://www.youtube.com/intl/en-GB/about/press/

[16] https://edu.gcfglobal.org/en/youtube/what-is-youtube/1/

[17] However, there is a di culty in the user interface where people reply as if they are replying to the comment above theirs, but this is actually not the case. This means that there is a reply structure between posts that, in terms of content, should be at the same structural level

[18] https://github.com/about

Commits are the way to make changes to a file that are shared across people in a project. Commits are 'pushed' to finalize the changes made within a branch.

Pull requests can be opened for merging branches. Making a 'pull request' means proposing the contributions a user has made for review so that the changes can be merged into a branch. The @mention system can be used in a pull request message for feedback requests.

4.3 Demonstrations

4.3.1 Demonstration Platform Implementations

Foobook

Pingstagram

4.3.2 Demonstration Scenarios

Scenario 1: Basic Foobook

This is the test used for **??**.

Scenario 2: Foobook with Malicious Users

Scenario 3: Foobook and Pingstagram

Scenario 4: Foobook and Pingstagram with Coordinated Users

Chapter 5

Create a New Platform

The main purpose of the RHPC_SMPLE toolkit is to allow the creation of a functionally faithful representation of a social media platform. The first step in this is the creation of a new platform object. In general, the practice is to create the platform abstractly at an upper level of implementation, and to do so in a way that allows a second customizable level. The upper level captures the known functionality of the platform, while the lower level permits the implementation of the user and platform behavior that is the object of some specific line of research. For example, the upper level may permit a piece of content to be 'liked,' while the lower level provides implementations that allow user agents to decide which content elements they will perform this action on.

This chapter gives instructions on how to do create the abstract objects at the upper level of the implementation. The overall sequence is:

1. Create all Header and Source Files
2. Create Definitions for Required Elements
3. Create Feed
4. Create Abstract Versions of Agent, User Agent and Content Agent

    Agent Type

    Action Type

    Relationship Type
5. Create a Payload Object
6. Create an Action Object

7. Create Required Network Elements

    Edge Information Objects

    Edges

8. Feed Query

9. Create a Platform Using the Above Objects

5.1 Step 1: Create All Header and Source Files

The platform code will require a version of all of the files shown in Table 5.1. Note that while some occur in pairs of header and source files, others need only header files.

Header Files	Source Files
Definitions.h	
Feed.h	Feed.cpp
Agent.h	
UserAgent.h	UserAgent.cpp
ContentAgent.h	
Payload.h	Payload.cpp
Action.h	Action.cpp
Network.h	
FeedQuery.h	FeedQuery.cpp
Platform.h	

Table 5.1: Files required to implement a new platform

Files can have different names than those shown here, e.g. customized for a particular platform. For example, “UserAgent.cpp” could be named “FoobookUserAccount.cpp”.

5.1.1 Step 2: Create Definitions for Required Elements

Creating de nitions is done by modifying *De nitions.h*. At this level, three elements must be defined:

1. Agent types
2. Action types
3. Relationship types (if applicable)

Additionally, this file must provide a forward declaration for the class that will be the platform, such as in the example below.

The example below defines two types of **agents**, *users* and *content* (termed ‘posts’). It also defines two types of **actions**: one begins a new thread and the other posts to a given thread. No relationships are defined in this example.

Sample Code 5.1: Adding Platform Definitions

```
/  Jamii_Definitions.h  /
#ifndef INCLUDE_JAMII_DEFINITIONS_H
#define INCLUDE_JAMII_DEFINITIONS_H

namespace jamii f

enum JamiiAgentType f
        USER_AGENT,
        POST_AGENT
g;

enum JamiiEventType f
        IDLE,
        POST,
        COMMENT,
        JAMII_EVENT_COUNT   // Used to determine the number/count
                            // of Enum elements for array access
g;

g // End namespace jamii


#endif /  INCLUDE_JAMII_DEFINITIONS_H  /
```

5.1.2 Step 3: Create the Feed Architecture

A **feed** is, fundamentally, a list of elements. It is similar to the ‘feed’ that a user of a social media platform provides via the user interface (for example, friends’ recent posts on Facebook).

In RHPC_SMPLE , a feed can also contain an *explanation* of why each element was placed into it. A typical definition of the feed begins by declaring the class that contains this explanation (if needed).

Example:

```
class JamiiFeedExplanation: public Explanationf
public:
        bool randomlySelected;

        JamiiFeedExplanation();
        JamiiFeedExplanation(const JamiiFeedExplanation& orig);

        virtual ~JamiiFeedExplanation();

        virtual std::string getRow();
g;
```

In the above example, the ***explanation*** is simple: a flag indicates whether or not the element was randomly selected.

Note that the second constructor is **required**. A copy instructor is needed because explanations are transferred among feeds and copied.

After the explanation, the **element** in the feed must be defined:

```
typedef FeedElement<JamiiAgentType, JamiiEventType,
    JamiiFeedExplanation> JamiiFeedElement;
typedef std::deque<JamiiFeedElement>::iterator JamiiFeedIterator;
```

Finally, the **feed** itself must be defined:

```
class JamiiFeed: public Feed<JamiiAgentType, JamiiEventType,
    JamiiFeedExplanation>f

protected:
        SimpleWeightedSelector  sws;

public:


        JamiiFeed();
        JamiiFeed(int maxSize);
        virtual ~JamiiFeed();

        void initiateWeightedSampling();

        JamiiFeedElement getOneSelectedWeighted();
```

```
g;
```

Note that this implementation provides some methods that are used to select elements from the feed.

5.1.3 Step 4: Create Abstract Versions of Agent, User Agent and Content Agent

As an agent-based model (ABM), agents are fundamental. In RHPC_SMPLE , there are two kinds of agents: there are agents that represent *users* (**user agents**) and agents that represent *content* (**content agents**).

Both of these derive from a single abstract class that is the basic **agent class** in the RHPC content and the fundamental 'node' element in the RHPC network.

Step 4.1: Create the Abstract Version of the Agent

At this level, the **agent class** needs only to be defined abstractly. Templatize the agent class to accommodate an action type that will not be specified until lower in the class hierarchy.

Example:

```
/  Jamii_Agent.h  /
#ifndef INCLUDE_JAMII_AGENT_H
#define INCLUDE_JAMII_AGENT_H

#include "socialnetwork_toolkit/SocialNetwork_Platform.h"

#include "Jamii_Definitions.h"
#include "Jamii_Feed.h"

namespace jamii f

class Jamii; // FORWARD DECLARATION

template<typename ACTIONTYPE>
class AbstractJamiiAgent: public SocialNetworkNode<JamiiAgentType,
        ACTIONTYPE>f

protected:
        Jamii                          jamii;
        boost::shared_ptr<JamiiFeed> myFeed;

public:
        AbstractJamiiAgent(repast::AgentId id, JamiiAgentType
                whichType, ptime dateCreated, SNP platform,
                SocialNetwork_AbstractElement
                <JamiiAgentType> creator = 0);
        virtual ~AbstractJamiiAgent();

    virtual void setCurrentRank(int currentRank);

    virtual std::string showName();
    virtual std::string writeName();

    void placeInFeed(SocialNetwork_AbstractElement
                        <JamiiAgentType> actor,
                SocialNetwork_AbstractElement<JamiiAgentType>
                        target,
                        SocialNetwork_AbstractElement
                                <JamiiAgentType> subtarget,
                        JamiiEventType action,
                JamiiFeedExplanation fe);

    virtual void setPlatform(SNP platform);

    int getCountOfFeedElements();

    virtual void selectTarget(ACTIONTYPE& action);

g;
```

The individual procedures that the agent implements are:

setCurrentRank(int):

showName():

writeName():

placeInFeed(actor*, target*, subtarget*, action, explanation):

setPlatform(platform*):

getCountOfFeedElements():

selectTarget(action&):

Step 4.2: Create the Abstract Version of the User Agent

An important aspect of **user agents** is that they represent ***user accounts***. This is not the same as representing ***users***. Users are people, and people can use multiple social media platforms (sometimes simultaneously!). User accounts, conversely, are platform-specific. They can perceive only things on that platform and can only take actions that the platform defines.

Most user functionality is defined at the implementation level. Below this level in the class hierarchy, so the class need not hold much functionality. In fact, class is used below only as a demonstration.

Example:

```
/  Jamii_User_Agent.h  /
#ifndef INCLUDE_JAMII_USER_AGENT_H
#define INCLUDE_JAMII_USER_AGENT_H

namespace jamii f

class AbstractJamiiUserAgent f

public:

        AbstractJamiiUserAgent();
        virtual ~AbstractJamiiUserAgent();

g;

g // End namespace jamii


#endif /  INCLUDE_JAMII_USER_AGENT_H  /
```

Step 4.3: Create the Abstract Version of the Content Agent

Content agents can be more robustly defined at this level than user agents can be.

In the example below, the content agent has been endowed with a way to collect ***Information ID*** values in the ***informationIDs*** set.

Example:

```
/  Jamii_Post_Agent.h  /
#ifndef INCLUDE_JAMII_POST_AGENT_H
#define INCLUDE_JAMII_POST_AGENT_H

namespace jamii f

template<typename ACTIONTYPE>
class AbstractJamiiPostAgent f

public:
        std::set<InfoID > informationIDs;

        AbstractJamiiPostAgent();

        virtual ~AbstractJamiiPostAgent();

        virtual void receiveAction(ACTIONTYPE& action);
g;

g // End namespace jamii

#endif /  INCLUDE_JAMII_POST_AGENT_H  /
```

The content agent is further specified by the ***receiveAction*** method. This method is **required** because it allows content agents to be acted upon by users.

The example below defines one critical piece of functionality, which is receiving the Information ID values and storing them in the set:

```
template<typename ACTIONTYPE>
void AbstractJamiiPostAgent<ACTIONTYPE>::receiveAction(
                ACTIONTYPE& action)f
        informationIDs.insert(
            action.payload.informationIDs.begin(),
            action.payload.informationIDs.end());
g
```

5.1.4 Step 5: Create the Base Class for the Payload

A **payload** is a package of data that is shipped with an action; an action will contain a payload, and the payload will be the ‘stuff’ that an action includes.

Generally, the payload object represents the genuine ***content*** portion of the content object. If a content agent contains elements like an ID, references to the user agent that created it, etc., it will also contain some content. This will be domain-specific but could include anything that is relevant in a given simulation: text, hashtags, abstract pieces of information, etc.

5.1.5 Step 6: Create the Action

As with a feed, an **action** can carry information that explains why the action was taken. Therefore the first component of an action is an *ActionExplanation*.

Example:

```
class JamiiActionExplanation: public ActionExplanation
        <JamiiFeedExplanation>f
protected:
        bool becauseOtherUsersLiked;

        std::vector<std::string> headings;

public:
        JamiiActionExplanation();
        JamiiActionExplanation(JamiiFeedExplanation feedExp);

        virtual ~JamiiActionExplanation();

        std::string getRow();
g;
```

In the above example, the only portion of the explanation is *becauseOtherUsersLiked*, which is a placeholder for more informative flags. Note that the *Action Explanation* extends, and therefore contains, the *FeedExplanation* defined above. Both of these are therefore extensions of **explanation**, which carries with it some inherent functionality (especially the ability to take one explanation and ‘add’ it to another by performing a logical ‘or’ on all boolean flags, etc.

Example:

```
template<typename AGENTTYPE, typename OUTPUT_TYPE,
        typename PACKAGE_TYPE>
class AbstractJamiiAction:
                    public SocialNetwork_Action<JamiiEventType,
                    AGENTTYPE, SocialNetwork_AbstractElement
                    <JamiiAgentType>, JamiiPayload,
                    JamiiActionExplanation,
                    OUTPUT_TYPE, PACKAGE_TYPE>f

public:

        AbstractJamiiAction(JamiiEventType event, AGENTTYPE
        actingElement = NULL, AGENTTYPE  targetElement = NULL);
        virtual ~AbstractJamiiAction();

        virtual std::string getOutputRepresentation(OUTPUT_TYPE
                filetype, bool writeOnlyFiltered = false);

        virtual std::string getDomain();

g;
```

```
template<typename AGENTTYPE, typename OUTPUT_TYPE,
        typename PACKAGE_TYPE>
std::string AbstractJamiiAction<AGENTTYPE, OUTPUT_TYPE,
        PACKAGE_TYPE>::getOutputRepresentation
        (OUTPUT_TYPE filetype, bool writeOnlyFiltered)f
    std::stringstream s;
    return s.str();
g
```

5.1.6 Step 7: Create Required Network Elements

Defining a network involves defining *ags*, *counters*, and *relationships*. The example below shows only counters. Flags are y/n and are used for things like "X created Y" (true/false). Counters are used to count events (e.g., "1,234 likes").

Below is an example of a basic counter that can count the number of Jamii events that occur between two agents.

```
class JamiiEventCounter:
          public SocialEventCounter<JamiiEventType>f
public:
         JamiiEventCounter();
         virtual ~JamiiEventCounter();
g;
```

Step 7.1: Edge Information Objects

Counters can use these to make **edge information** objects. An edge information object stores a directed edge from two agents (e.g., from A to B). This is used in conjunction with flags/counters/relationships to indicate how A is related to B (e.g., “A has ’liked’ B 6 times).

To implement this, extend the appropriate class:

```
    public SocialNetwork_CountEdgeInformation<JamiiEventType,
         JamiiEventCounter>f
         friend class boost::serialization::access;

public:
         JamiiEdgeInfo();

         virtual ~JamiiEdgeInfo();

         template<class Archive>
         void serialize(Archive& ar, const unsigned int version)f
                  SocialNetwork_CountEdgeInformation<JamiiEventType,
                  JamiiEventCounter>::serialize(ar, version);
         g
g;
```

Step 7.2: Edges

After implementation, you can create an **edge** that contains the ***edge Information*** object. This allows declaring a formal interface that can capture the appropriate semantics for the platform you are creating.

Note that /emphgetNumPosts is not intrinsic to RHPC_SMPLE , but rather is defined by the fact that in Jamii there are 'Posts'.

```
template<typename AGENTTYPE, typename EI> // EdgeInfo
class JamiiEdge : public repast::RepastEdge<AGENTTYPE>f
private:
    EI _edgeInfo;

 public:
    JamiiEdge();
    JamiiEdge(AGENTTYPE  source, AGENTTYPE  target);
    JamiiEdge(AGENTTYPE  source, AGENTTYPE  target, double weight);
    JamiiEdge(AGENTTYPE  source, AGENTTYPE  target, double weight,
    EI edgeInfo);

    JamiiEdge(boost::shared_ptr<AGENTTYPE> source,
        boost::shared_ptr<AGENTTYPE> target);
    JamiiEdge(boost::shared_ptr<AGENTTYPE> source,
        boost::shared_ptr<AGENTTYPE> target, double weight);
    JamiiEdge(boost::shared_ptr<AGENTTYPE> source,
        boost::shared_ptr<AGENTTYPE> target, double weight,
        EI edgeInfo);


    EI& getEdgeInfo();
    void setEdgeInfo(EI edgeInfo);

    // Querying
    int      numPosts()f
                                    return _edgeInfo.getCount(POST); g

    bool     any()f
                                    return !(_edgeInfo.isEmpty()); g

    // Updating
    void          logEvent(JamiiEventType event)
                  f _edgeInfo.recordEvent(event); g


        int       getTotalEvents()f
                    return numPosts();
```

```
          g

g;
```

5.1.7 Step 8: Create a structure for a Feed Query

A **feed query** is a class that contains the information needed for the platform to execute a query. In other words, it is a a set of criteria used to return a collection of content objects.

The feed query can be a simple structure.

Example:

```
/  Jamii_Feed_Query.h  /
#ifndef INCLUDE_JAMII_FEED_QUERY_H
#define INCLUDE_JAMII_FEED_QUERY_H

namespace jamii f

// Feed query types
enum JamiiFeedQueryType f
          RETRIEVE_RANDOM_POSTS,
          JAMIIFEEDQUERYTYPE_META
g;


class JamiiFeedQuery f

public:
          JamiiFeedQueryType type;

          int countOfRandomPostsToRetrieve;

          JamiiFeedQuery(JamiiFeedQueryType t =
                  JAMIIFEEDQUERYTYPE_
```

```
#endif /  INCLUDE_JAMII_FEED_QUERY_H  /
```

5.1.8 Step 9: Create a Platform Using the Above Objects

A platform must:

1. Allow creating and updating **user agents** and **content agents**

   This includes setting local pointers

2. Allow responding to **feed queries** by providing feeds that include relevant items
3. Allow performing **actions** from user accounts onto user or content nodes

At this level, ***creating and updating*** are not yet implemented. Implementation will happen at the lower level. Additionally, the responses to queries are not defined at this level; this is instead application-specific.

Instead, the actions to be performed for the **API** (Application Programming Interface) are defined at this level. The API is a list of functions that a piece of software (or a social media platform) can perform.

For Jamii, the only actions are 'post' and 'comment' (see the definitions above). The relevant sections in the platform interface are:

```
// API Functions
void doAction(ACTIONTYPE action);
void api_startThread(ACTIONTYPE action);
void api_comment(ACTIONTYPE action);
```

In this case, the first is generic and should pass to the appropriate semantic operation based on the action passed. The 'startThread' and 'comment' actions represent 'posting' and 'commenting.'

Several other components of the platform may be defined at this level; look at the examples provided for details.

Chapter 6

Extend a Platform to a Demo

The previous chapter outlined the steps required to use the RHPC_SMPLE toolkit to create a social media platform at an abstract level, capturing the basic functionality of the platform at the upper level while leaving the specifics for implementation at a lower level. These specifics could include, for example, agent behavior representing user accounts that interact with the platform's functionality in ways that are customized to the research question being explored.

This implies a second step: the implementation of this lower level, so that the toolkit can be used to run simulations of online behavior. This chapter presents the steps necessary for running simulations.

6.1 Hierarchy of Functionality

Before discussing the step-by-step functions, a conceptual diagram of a relationship is presented to show the relationships among the classes declared in the previous chapter and the classes that will be discussed and implemented here. Figure **??** shows these relationships.

6.2 Step-by-Step Instructions

Below is a list of instructions. Each step is explained in the next section.

1. Create all the Header and Source Files (empty)
2. Create a makefile that compiles all source files

3. Populate the Header Files

    Definitions

    Agent Package

    Custom Action

    Agent

    Content Agent

    Behavior Selection

    InfoIDSelector

    User Agent

    FileWriter

    Feed Query

    Platform

4. Populate the Source Files

    Package

    Action

    Agent

    Content

    Behavior

    User

    Feed Query

    Platform

6.2.1 Create all the Source and Header Files

A good first step is to simply create the (empty) files as listed in the table.

Header Files	Source Files
Definitions.h	
Action.h	Action.cpp
Agent_Package.h	Agent_Package.cpp
Agent.h	Agent.cpp
ContentAgent.h	ContentAgent.cpp
UserAgent.h	UserAgent.cpp
BehaviorSelection.h	BehaviorSelection.cpp
FeedQuery.h	FeedQuery.cpp
FileWriter.h	
InfoIDSelection.h	
Platform.h	

Table 6.1: Files requred to implement a platform

6.2.2 Create the Definitions for File Output Type

Here you should define a set of types that will be used to score agents when their order of retrieval is being determined.

If any file output is needed, then a definition of output type is required. See Chapter 9 for more information.

See Twitter_De nitions_xamples.h for an example of both of these.

6.2.3 Create the Agent Package

The agent package should include all information needed to construct an identical copy of an agent.

See Twitter_Agent_Package_xamples.h for an example of both of these.

6.2.4 Create the Abstract Agent, the User Agent, and the Content Agent

The agent will inherit from the abstract agent, and will include any functionality that a general agent on this platform (whether user or content) should have. It will be templated by Action (from the platform level) and Score Type from the definitions at this level. *See Twitter_Agent_xamples.h for an example.*

User Agent

The user agent will descend from the abstract user agent at the platform level and the agent at the demo level, and must implement selectBehavior, selectTarget, and selectInfoIDs methods. *See Twitter_Agent_User_xamples.h for an example; this example omits the inheritance of the demo level agent, but it is not needed for the demo.*

Content Agent

The content agent will descend from the abstract agent at the platform level and from the agent at the demo level.

The content agent must implement two significant methods:

> *receiveAction* This must process an action when this content target is acted upon. This includes any modification to the state of the target and any changes to the action.
>
> *getSelfPointer* This must return a pointer to self; it is used in specific situations where the pointer must be to the lowest child class but is generated by the parent.

This structure allows the user agents and content agents to inherit demo specific behavior but avoids a diamond inheritance pattern with the objects at the platform level.

See Twitter_Agent_Conversation_xamples.h/.cpp for an example.

6.2.5 Create the Custom Action

Description the custom action must be able to perform two actions:

> Return an output representation (see Chapter 9 for examples)
>
> Return a new agent package. This is a package that represents all of the information needed to create a new content agent if the result of the action is that a new agent is to be created.

See Twitter_Action_xamples.h/.cpp for an example.

6.2.6 Create the Feed Query

The Feed Query must carry all of the information needed to retrieve a collection of content elements from the platform. This generally includes a Feed Query Type, and additional information that is used to define the request. *See Twitter_Feed_Query_xamples.h for an example.*

6.2.7 Create the Concrete Implementation of the Platform

The main roles for the platform are:

1. Creation and updates of agents
2. Responding to Feed Requests

Creation of Agents

There are eight significant methods that must be implemented for this functionality:

getPackage Returns an agent package for an *existing* agent.

updateAgent (Package only): Updates an agent; the agent is identified in the package. This generally passes through to the next method

updateAgent (Package and Pointer): Updates a specific agent using the data in the package

provideContent This is a Repast HPC standard; given an 'agent request', populate a collection of agent package information

providePackage This is also a Repast HPC standard; given a specific agent, add that agent's information to a collection of agent package

createNewAgent This should create a new agent and return the pointer

createAndAddNewAgent This should create a new agent *and* add that agent to the context. It generally works by calling the createNewAgent method, but is also responsible for managing that agent's existence within the platform.

setLocalPointers When a new agent is created (or, in parallelization, when an agent is copied from another process) it may need pointers to local instances of objects. Pointers cannot be packaged and sent; instead, some mechanism for finding the equivalent object on the local platform must be made. Usually this method is also used for newly created agents.

Some of these functions are driven by the needs of parallelization, but because they are determined by Repast HPC they must be implemented whether parallel runs are needed or not.

See Twitter_Platform_xamples.h/.cpp for an example.

6.2.8 Step 8: Create the Behavior Selector

This is a class that takes an agent and returns one or more actions that the agent selects to undertake. *See Twitter_BehaviorSelection_xamples.h/.cpp for an example.*

6.2.9 Step 9: Create the InfoIDSelector

This is a method of selecting a topic to be discussed in the content. In the example *See Twitter_InfoIDSelector_xamples.h/.cpp for an example that achieves this entirely by typedef statements*

6.2.10 Step 10: Create the FileWriter

See Chapter 9.

Chapter 7

Create Scenarios

Once a platform has been created, it is necessary to create the architecture of a simulation around it; in terms of functonality, this includes such things as initializing a population of agents, scheduling actions for the agents to perform, creating a collection of external events to which these agents will respond, etc.

The coding structure that does this in RHPC_SMPLE is the `Scenario`. Scenarios are containers objects that store instances of platforms as well as other elements of the outside world, and manage the operation of these elements. Scenarios are like the 'sheep dog' of platforms: the elements required to simulate a social media platform, including the platform itself and aspects of the world in which it exists, are assembled and managed by the scenario.

Scenarios provide a means of exploring potential futures (rather than just one future) by presenting outcomes of potential decisions. Because of this, scenarios are used for trend prediction, such as with social media. This can work because scenarios represent the world in which platforms exist. Incorporating other elements of the external world into a scenario as exogenous data sources can help to see realistic potential futures.

7.1 Creating Scenarios

Building scenarios involves coding an object that extends a scenario class from the RHPC_SMPLE toolkit. A class has four discrete sections to it:

1. Construction
2. Initialization

3. Scenario Operation
4. Teardown

Examples of these steps can be see in the demos provided in the xamples directories. Note that it is common to create a parent class that captures the generalized functions of a scenario, and then child classes th provide the platform-specific details; this is followed in the demo code, with *Base_Scenario.h* and *SimpleTwitterScenario.h*.

7.1.1 Construction

There are six steps to the construction step of scenario creation. The constructor needs to set the rank value, collect any relevant global properties, create an instance of the platform, initialize any output files, create instances of collections of exogenous data, and create and add any subscenarios. Each of these requirements is explained below.

Set the Rank Value

The constructor sets the rank using the following code:

```
          rank = world >rank();
```

Collect any Relevant Properties

Scenarios can use properties (from properties files or the command line; see Chapter **??**). The scenario can also use properties with default values that are used if no specification of the property is present in the properties file or the command line. If the properties were not specified originally, they are added to the properties collection with their default values.

There is an important consideration with default properties: Any auditable record of property values—that is, a file that is written that contains the values of the properties used for a specific simulation run—must be written after any default properties are specified. Changes to properties that are made after the audit file is written do not, of course, appear in the record, rendering the record incomplete or inaccurate. To prevent this, the *writeMetadataFile* function of the scenario object sets a flag that makes the use of default values impossible; any call to *setPropertyDefault* or *getPropertyWithDefault* after this flag is set will cause an error.

Create an Instance of the Platform

This is achieved simply; here, ‘world’ means a connection to the MPI Communicator object and ‘this’ is the scenario itself.

```
foobook = new Foobook(world, this)
```

Initialize any Output Files

See 9

Create Instances of Collections of Exogenous Data

This is an optional but common step: if the simulation is relying on exogenous data (like lists of event counts per step, or data that informs the agents’ actions), it can be initialized here.

Create and Add any Subscenarios

Commonly, a scenario will be a specification for one platform; it can manage everything that is needed for that platform to be used in a larger simulation. However, simulations often use multiple platforms simultaneously. To achieve this, the ability to nest scenarios inside one another is used. The strategies are to create the overarching scenario that represents the entire simulated world, to create the separate scenarios that represent individual platforms, and then to add the platform scenarios as **subscenarios** to the overarching scenario.

If this is done properly, RHPC_SMPLE will automatically handle both initialization and operation of the overarching scenario and all its subscenarios. At the toolkit level, functionality is defined that calls functions on a scenario and its subscenarios, so that actions cascade downward. For example, when the call is made to *initUserToUserLinks*, the parent scenario first calls the equivalent function on its children, and then calls it on itself. This means that the initialization is triggered automatically for all levels.

Subscenarios can be added using the following example:

```
std::string rdPropertiesFileName =
   properties >getProperty("rd.subscenario.properties.filename");
repast::Properties  rdProps =
   new repast::Properties(rdPropertiesFileName, world);
propagatePropertiesDownward("RD", rdProps);
```

```
rdScenario =
   new CP6_RedditScenario(world, model, rdProps);
redditFileWriter =
  new reddit::Reddit_FileWriter(reddit::Reddit_FileOutputTypes::
  JSON, globalJSONOutput_ptr);
rdScenario >reddit >fileWriters["REDDIT.JSON"] =
  redditFileWriter;
addSubscenario("RD", rdScenario);
```

The first line determines the name of an extra properties file, which is loaded in the second line. The third line propagates the properties of the parent scenario downward if those properties begin with the “RD__” prefix. The fourth line creates the scenario; the fifth and sixth lines create a file writer using a global pointer and add it to the child scenario. The last line adds the subscenario to the parent scenario.

Properties can be sent downward from parent scenarios to child scenarios. This allows the user to specify properties using the different methods discussed in 10 and to send specific property values to specific subscenarios.

7.1.2 Initialization

Initializing the scenario means loading the data and creating the initial state for the simulation. This includes primarily nodes and the links among them, or:

Nodes

  Users

  Content

Links

  Content ! Content links

  User ! Content links

  User ! User links

  Content ! User links

initScenario()
initUserAgents()
initContentAgents()
initContentToContentLinks()
initUserToUserLinks()
initUserToContentLinks()
initContentToContentLinks()

Table 7.1: Scenario Initialization Functions

The initialization functions fall into three categories:

1. Initialize the overall scenario
2. Initialize the individuals nodes

   user

   content

3. Initialize the links between nodes

The list above shows the sequence in which the methods are executed. All are optional. They can be omitted if they do not apply in a specific scenario, but generally, ***initScenario*** and at least either ***initUserAgents*** or ***initContentAgents*** will be needed.

An important aspect of this sequence is that the ***initScenario*** function typically collects data needed for the subsequent functions. One key example has to do with agent identifiers.

Typically, files store agent identifiers as character strings. However, RHPC_SMPLE does ***not*** use these internally. Once the simulation is running, it is difficult for one agent to interact with another on the basis of its identifier. This is deliberate and derives from the core agent-based modeling tenet that the model stores relationships among agents, and the perception that any agent has of other agents in its world is mediated through these relationships.

Through these relationships, it is possible for an agent to ask, "What agents are near me in space?" It is also possible to ask, "What agents are connected to me via the network?" When these queries are answered, the answer is a list of references to the other agents. However, it is difficult to ask, "Where is agent XZY?" The reason for this difficulty is that in the absence of an existing relationship between the original agent and agent XYZ, there is no reason the original agent ***should*** be interacting with XYZ.

One exception is provided by the toolkit, which is the selection of agents at random—a common ABM technique. The overall query functionality of the platform is another example: in theory, an agent can ask for a specific agent with which to interact, but only if it has received information about that agent. More generally, the query will ask for agents meeting a range of criteria.

Because of this structure and limitation, ***the identi ers in the original data les are not used by the toolkit.*** This is especially important during initialization, when the initialization files may say that agents (identified by strings) have a relationship, but these identifiers are lost when the load occurs. Instead, a sequence like this must be followed:

1. During ***initializeScenario***, make an empty map of identifiers and pointers to agents.
2. During ***initializeUserAgents*** and/or ***initializeContentAgents***, make agents that correspond to those identifiers and store the pointers.
3. During ***initialize...Links()***, use the maps to retrieve pointers, and from these use the built-in RHPC_SMPLE functions to establish connections among the agents.

These maps can be destroyed after initialization; this is often done in the ***cleanupInit()*** method, which is called automatically after scenario initialization.

7.1.3 Scenario Operation

There are two aspects to the creating the scenario operation. First, there is ***performExogenousEvents***, which updates the world. Second, there is ***performAgentActions***, which responds to the world.

Generally, these methods can simply be implemented, and the toolkit will execute them each step. Subscenarios' implementations will be called along with the parent's.

7.1.4 Teardown

Scenario destructors are optional.

Chapter 8

Explanations

8.1 Overview

The RHPC_SMPLE platform is designed to allow creation of virtual social media platforms and populations of simulated users. The purposes of simulations can vary, but there is generally a goal of understanding how populations of user-agents interact and evolve given the affordances of a particular platform and the strategies being pursued by both platforms and users. In a simulation, both the platforms and the users are agents.

One common simulation strategy is to run multiple variations of the simulation where each variant is determined by a set of parameters, and the set of runs is given by all the possible (or interesting) combinations of these parameters. If the simulation includes stochasticity, then each distinct combination typically needs to be run multiple times in order to establish the range of variance in the results. The set of results across all runs can be subject to statistical analyses to show the likelihood of specific outcomes and the relationships of these outcomes to the parameter values that determine them.

Such a high-level approach is appropriate for many circumstances, but a complementary approach can also be useful. Rather than take a statistical survey of results, insight from the model can be gained from a 'deep dive' into the operation of the simulation.

To facilitate this, RHPC_SMPLE provides a structure called an `explanation`; instances of explanations can be inserted into the payloads that are exchanged between the simulated platforms and the simulated users. The explanation object can be customized to record specific decisions, such as why a particular piece of content was inserted into a specific user's feed. Decisions can then be recorded and inspected.

This level of inspection is one of the things that differentiates the modeling approach for which RHPC_SMPLE was created from other approaches, such as machine learning or A/I approaches. By providing a window into why things are happening into the simulation, the simulation can be used in a way that the 'black box' of a machine learning approach cannot.

Naturally, the explanations are not only useful for 'deep dives'. It is also possible to collect data at a statistical level from explanatory outputs.

8.1.1 Implementation

The main technical advantage of the explanation object is that it is additive: two explanation objects can be merged, and the result is a record that includes all of the explanations that were in the original pair. This allows for different elements to write to the explanation and for the explanation to accumulate reasons as it moves through the decision pipeline.

Using the explanation object requires extending the explanation base class by adding fields that indicate the different kinds of explanations available to the platform and users. To maintain functionality, extensions to the explanation object must be done in a specific way.

8.2 Feed Explanations

Feed explanations allow the platform to record the reasoning used to place specific pieces of content into users' feeds. Examples of reasons might include:

The content was created by a user that the recipient user follows.

The user queried the platform for content with a specific identifier.

The content was popular among other users the platform considers similar to the recipient.

8.3 Action Explanations

Action explanations indicate the reasons that a simulated user elected to perform a specific action on a piece of content. Examples of reasons might include:

The user disagreed with the content.

The user wanted to engage more with the creator of a piece of content and responded.

The user was attempting to disrupt a conversation.

For an example of both kinds of explanations, see *Twitter_Feed.h* and *Twitter_Action.h*.

8.4 Writing Explanations as Output

Writing output from explanations follows the same procedure as writing for actions. Typically, an action responds to a *getRepresentation* request (see Chapter 9) with a string that describes the action taken in terms of the actor, the target content, and information about the nature of the content (i.e., *payload*, see Step 5 of Chapter 5). This same approach can accommodate information from the explanation.

One approach is to include explanation information in the same output representation as all of the other information from the action. It is also possible—and sometimes preferable—to direct explanatory output to a separate file. To do this requires creating a different response to the *getRepresentation* request and then creating a new FileWriter that requests that kind of response.

Chapter 9

Output

9.1 Overview

Writing to les to collect simulation data is a key operation, as it represents the way in which simulation output becomes data for analysis and the way the simulation's results are made visible for review and use. A simulation that produces no data is of little value.

The RHPC_SMPLE toolkit provides a built-in mechanism for writing output to files. The core of the framework is the *Action* object, which specifies a social media action undertaken by a user agent, and a *FileWriter* object, which directs output of a specific type to a specific file.

To make this work, an action should be able to respond to a request via *getOutputRepresentation(type)*, where 'type' identifies the kind of output file. The action's response is thus tailored to a type of output file. These 'types' are defined in code, and each type refers to a specific output format. For example, one may have three formats:

A csv file in which the actor and platform-specific action alone are listed

A csv file in which actor, action, timestamp, and items from the payload are listed, each item on a separate line

a JSON file in which the actor, action, timestamp and payload are listed, with payload elements as a JSON array

When a FileWriter is created, it specifies the output type that will be used to collect the actions' responses. It also specifies the destination file to which it points. This can be unique to a single FileWriter, or, alternatively, a single file for output that is shared among multiple FileWriters.

9.2 How to Implement Writing to Files

Here is a list of procedures for **writing to les**:

Step 1: de ne the kinds of output les in a *de nitions* le.

Example:

```
enum Twitter_Xamples_FileOutputTypes f
        TWITTER_XAMPLES_FILEOUTPUT_JSON,
        TWITTER_XAMPLES_FILEOUTPUT_CSV,
        TWITTER_XAMPLES_FILEOUTPUTTYPES_META
g;
```

This defines one format (JSON). The last entry is only used to indicate how many entries the enum contains.

Step 2: de ne the *getOutputRepresentation(type)* method in the *action*

The *Action* element should be defined to have a *getOutputRepresentation* method that takes one of the enumerated types as an argument. The only requirement of this method is that it takes the type as an argument and returns the text that should be written to the file. This often is a single line (e.g. a line for a CSV file), but can be more than one line if the output formate requires it. The closing end-of-line marker should be included.

Step 3: create an instance of the lewriter that uses the kinds of output les in the *de nitions* le

Typically this is done during the scenario constructor. A common pattern is to use a typedef in the platform's include file, such as:

```
typedef TypedFileWriter<Twitter_Xamples_FileOutputTypes>
        Twitter_FileWriter;
```

Then in the scenario constructor, all that is needed is:

```
Twitter_FileWriter   fileWriter_json
      = new Twitter_FileWriter(Twitter_Xamples_FileOutputTypes::
           TWITTER_XAMPLES_FILEOUTPUT_JSON,
           w_outFilename + ".json");
```

Notice that it is passed the output file type that it will use. If multiple output types are needed, multiple instances can be created.

Step 4: during scenario initialization, add this lewriter to the collection of lewrites in the scenario

The platform will have a collection of file writers; the newly created instance should be added to this collection. The collection is a map, and each entry gets a (unique) name, e.g.:

```
         twitter >fileWriters["TWITTER.JSON"] = fileWriter_json;
```

Step5: call *writeAction* in the platform

If the structure above is followed, calls to the platform API will automatically call a function that cycles through all available file writer instances and writes the action to each one.

9.2.1 Using multiple writers to a single file

In some cases a single simulation will write output from multiple sources (e.g. multiple platforms) to a single output file. This can be easily accommodated by creating the output stream independently, then creating the file writer instances using the alternate constructor that takes a stream as an argument. When this is done, the stream is not managed by any of the individual writers and must be closed independently.

9.2.2 Pausing

All file writer instances inherit the ability to move from an active to a 'paused' state and back. While in the paused state, no output is written. This is useful for processing certain forms of pre-simulation activity, as well as other cases.

Chapter 10

Launching a Simulation

10.1 Specifying Properties

It is almost always necessary to run multiple variations of a simulation in which crucial parameters or features are varied; this is done to explore multiple possible outcomes. In RHPC_SMPLE, this is done by specifying **properties**. Properties are things like parameters (e.g., coefficients) or switches (e.g., alternative algorithms) that change the way the simulation operates.

For example, consider the property ***agent speed***. For variation, run a scenario three times with agent speed set at 1, 2, and 3. The result of this is doing multiple runs with multiple property settings.

There are two ways to specify properties for running a simulation:

Specify properties in a properties file. This can include a main file and/or a file for each platform.

Give the name and the value of a property when the simulation is run from the command line. This can include prefixing properties.

The properties le is required, but specifying individual properties on the command line when the simulation is run is optional. If used, the values on the command line override any that are in the properties file. Note that the scenario can also define properties with default values if they are not present in the properties file or the command line. However, the scenario should also set a flag for making default properties invalid after the collection of properties is written to the metadata file. The importance of this flag is that after properties are written to the metadata file, no changes can be made to the metadata file.

Specify Properties in a Properties File

A common way of configuring the simulation to run with alternative sets of properties is to specify values in a main file and/or a file for each platform. In other words, each platform can be controlled through the main file, which is the ***multi-properties (mp) le***, or individually through different files. The values in the main file are read by RHPC_SMPLE into the simulation, thereby determining how it runs.

Give the Name of a Property for a File

There is an additional way to provide parameters to individual simulation runs: it is possible to override properties by adding to the *command line*. It can be useful to be able to run the simulation from a command line (or within a script) and to explicitly specify parameters without modifying a properties file. This way, properties prefixes are included. Appropriate prefixes are given in the example below.

Consider that the original property name to be given is *speed.of.agents*.

One of several platforms to specify from the command line is *Twitter*. Twitter is specified with with prefix *TW*. So, for Twitter, use *TW__speed.of.agents*. (***NOTE***: two underscores are used.)

Likewise, for Jamii, use prefix *JM*: *JM__speed.of.agents*.

To specify *all* platforms from the command line, use prefix *MP*:
MP__speed.of.agents.

A common use of this approach is to keep a file with *default* properties, specifying any properties that vary from the default when the simulation is invoked.

Bibliography

[1] 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*, vol. 1, no. 1, pp. 1–26, 2013.