Toward Orchestration of Complex Networking Experiments

Alefiya Hussain *USC/ISI*

Prateek Jaipuria[*] *Hulu Networks*

Geoff Lawler *USC/ISI*

Stephen Schwab *USC/ISI*

Terry Benzel *USC/ISI*

Abstract

Experimentation is an essential tool for developing networked and distributed systems. However, it is inherently complex due to the concurrent, asynchronous, heterogeneous, and prototype-based systems that must be integrated into representative scenarios to conduct valid evaluations. This paper offers a retrospective on the development and use of MAGI, an orchestration tool, that translates an experiment specification into an execution on an emulation-based testbed with high-level directives for message passing, remote process execution, and failure tracking, for conducting large and complex experiments. The MAGI tool has been used for more than seven years in a variety of experiments, including undergraduate education, anonymous communication, cyber-physical systems, and attacker-defender games on the DETER testbed. We hope the insights and takeaways learned from using our tool will aid in developing the next-generation experiment management tools.

1 Introduction

Over the past two decades, many emulation-based and internet-based testbeds have been developed to enable realistic evaluations of networked and distributed systems. However, the process of evaluating and tuning alternative designs and implementations of distributed systems is intrinsically complex. The complexity is mainly due to the high levels of concurrency and asynchrony of such systems and the practical obstacles that developers face in evaluating their systems with representative scenarios. Creating representative evaluation scenarios requires the concurrent execution of traffic generators and monitoring systems along with the prototype distributed system. The sequence of steps required to create such representative scenarios on a testbed is known as *experiment orchestration*. Experiment orchestration grows increasingly challenging as the scale and complexity of networked systems grow [5, 24].

Experimentation allows developers to understand the various dynamic system properties and tune their systems before transitioning into the real world [32]. Experimentation with security properties of distributed systems is particularly important and particularly challenging: important because in the presence of attacks, distributed systems can exhibit a wide range of unanticipated behaviors; and challenging in that it can be difficult, costly, and time-consuming to reproduce all possible configurations of a distributed system, under all possible usage scenarios, and under all possible environmental conditions, in the testbed.

The terminology of experimentation is not universal. In this paper, we define an experiment to be a list of steps for an individual run of the distributed system. These steps include a specific set of tuning parameters for the system components, the configuration parameters for background traffic components, and the deployment and execution of the components to the testbed nodes and network elements. Experiment orchestration is thus the process of executing the sequence of steps that define the experiment. For statistically sound results, a series of identical experiments are typically executed. Additionally, different experiments are required to optimize the performance of the system, for example, by varying a tuning parameter across a range of values and analyzing the performance of the system.

Many networking experiments are meso-scale representations of an internet or enterprise network. They attempt to recreate the network conditions in terms of topological construction and application traffic mixes. Experiment complexity comes from several sources. In some experiments, the complexity results from a unique set of required conditions, for example, the volume of traffic on a particular link in the topology needs to meet a threshold in order to evaluate the performance of a server under stress. If the generation of the traffic is not done correctly, the experiment may unknowingly produce invalid results and not succeed. In many cases, experiment complexity is due to the integration of technologies from different sources. For example, when teams collaborate on a project, team members bring different technologies to the

[*] Work done while at USC/ISI

experiment or use off-the-shelf components. The integration and orchestration of diverse technologies benefit immensely from automated experiment orchestration. Lastly, in research or educational settings, an experiment is repeated to recreate the results by another researcher or a student. In such cases, some form of experiment orchestration is crucial to enable the execution of the experiment.

A large and complex experiment on a testbed can be viewed as a single distributed system with multiple components that must run concurrently to generate traffic, monitor, and analyze performance. Most experimenters develop only a subset of the components of the experiment, such as a network defense system, and want to evaluate their defense system with a wide range of traffic and network configurations developed by other researchers. In this paper, we discuss the MAGI (Montage AGent Infrastructure) experiment orchestration tool that has been widely used over the last seven years to evaluate distributed systems on the DETER testbed. MAGI is part of a larger collection of experimentation tools, including an experiment lifecycle manager and repository [21, 22]. In Section 2 we provide a survey of current orchestration techniques and tools used in network and emulation testbeds. Where applicable, we discuss how MAGI built on the ideas of previous generations of orchestration tools and extends their capabilities.

In Section 3, we discuss the key components of the MAGI tool that enable experiment orchestration and graceful error handling. In a nutshell, an experiment starts as a conceptual model that is translated into a specification and orchestrated on the testbed by the orchestrator, node-based daemons, and agent modules. The specification combines high-level directives such as *events* and *triggers* to enable remote execution, coordination, and control in the experiment.

In Section 4, we present illustrative examples that document our experience of using the MAGI tool for experimentation. The examples illustrate the applicability of MAGI in many environments, such as a senior-level undergraduate course in networking, multi-party games, for safe and anonymous communication on Tor networks and the evaluation of distributed control algorithms in cyber-physical systems. We summarize our key takeaway from each experience. In Section 5, we evaluate the overhead of the orchestrator and agent daemons and show that it is minimal for most experiments. We then outline a path forward in next-generation testbeds. Finally in Section 6, we summarize the retrospective takeways from our experience with the development of MAGI, and attempt to provide guiding principles for future development of such tools.

The scale and complexity of distributed systems experimentation will continue to grow with the seamless integration of virtualization technologies and cloud-based testbeds [4, 9, 17]. Experimentation tools are typically developed in conjunction with a testbed and have an overlapping usage and retirement arc [7]. The DETER testbed has been operational for almost two decades and is now metamorphosing with the next generation testbed technologies [17]. This paper is thus a retrospective on the development and use of an experiment orchestration tool on the DETER tested. The goal is to capture our experience and share some key insights, that the authors believe, were critical to the success of MAGI and use them to develop an orchestration mechanism for the next generation virtualization-based testbeds. We hope it will initiate discussion and spur the development of experimentation tools for distributed systems.

2 Related Work

In this section, we briefly discuss a few tools and testbeds used for distributed system experimentation, along with insights into how they informed the design of our orchestration tool.

Shell or ssh-based scripts: Many experimenters design and execute experiments using a sequence of shell commands to execute programs on the experimentation nodes. We found that structure of the scripts fall into two major categories: some experimenters develop specialized scripts for each behavior in the experiment, transfer the scripts to the different nodes, and then execute the scripts on the nodes; other experimenters choose to develop a master script that remotely executes commands on the individual experiment nodes, using ssh-based tools, such as shremote [28] and the Python fabric library [19]. Shremote is an ssh-based tool for the timed execution of remote commands. It has a configuration file that allows running commands on remote machines, verifies success, and gathers log files for post-processing. Fabric fabfiles are Python programs that execute arbitrary commands on subsets of machines identified with Python decorators. The use of decorators and a programming language enables writing flexible configurations and control schemes, but limits the amount of feedback that can be received from the experiment. In both cases, processes need to be terminated using a OS *kill* command that could lead to sudden EOF for file processing or a TCP-RST for open connections. Such techniques limit the level of control and error handling in large scale experiments and can lead to hard-to-track failures that may manifest as incorrect analysis and results. However, this continues to be the most popular form of experimentation on testbeds [6, 20]

Ansible: Ansible is a configuration management (CM) and orchestration tool that leverages built-in modules to perform tasks on remote systems [3]. Ansible is agent-less with declarative semantics and is used by experimenters to describe the configuration of the nodes in a playbook. It is limiting when conducting with experiments that require expressing procedural complexity. Under the hood, Ansible uses ssh-based commands to configure the nodes; hence, any complex experiments that go beyond a simple list of tasks, such as configure, start, wait, and collect results, are difficult to express. Although we primarily discuss Ansible in this paper, there are several Ansible-like environments for configuration of systems such as Salt [33], TerraForm [35] and Chef [10]. Although, these systems cannot be effectively used for complex experimentation, they provide a reliable solution to deploy software to experiments.

Emulab Testbed Tools: The Emulab testbed was the first emulation-based testbed that allowed experimenters to run short experiments on networking protocols and systems [37]. Experimenters were able to configure nodes and topologies on demand with emulated network links between their nodes. The early version of the Emulab testbed had a tool, `tevc`, that enabled scheduling event sequences in a ns-2 topology description file or interactively using the command line client [37]. Later, the tested provided the Emulab Experimenters Workbench [14] with support for experiment versioning, cloning via templates, and archiving. These capabilities support pre-packaged experiments, and are useful features for sharing, but did not provide any additional support for experiment orchestration and management. The Emulab testbed was extended to include a ProtoGENI testbed to support federation of resources and incorporated into the GENI testbed [24]. More recently, it has added features to support cloud-based application evaluation under the aegis of CloudLab.

PlanetLab Testbed Tools: Althought PlanetLab officially retired in May 2020 [29], at its peak it supported more than 1300 nodes worldwide, and provided a sophisticated set of experimentation tools whose features were critical to the success of the testbed within the community [1]. Plush was a toolkit for distributed experiment configuration, management, and visualization. Plush provided a Nebula-based workbench through which users could request testbed resources, configure them with the required libraries and software, and view a runtime visualization of the experiment [2]. To manage the concurrency, Plush provide two synchronization primitives, predecessors and barriers. Predecessors allow the ordering of processes locally on a node; barriers enable the ordering of processes globally in the experiment. The Plush primitives, however, did not have the expressiveness to define complex dependency constraint in the experiment. The PlanetLab testbed was incorporated into the GENI testbed [6].

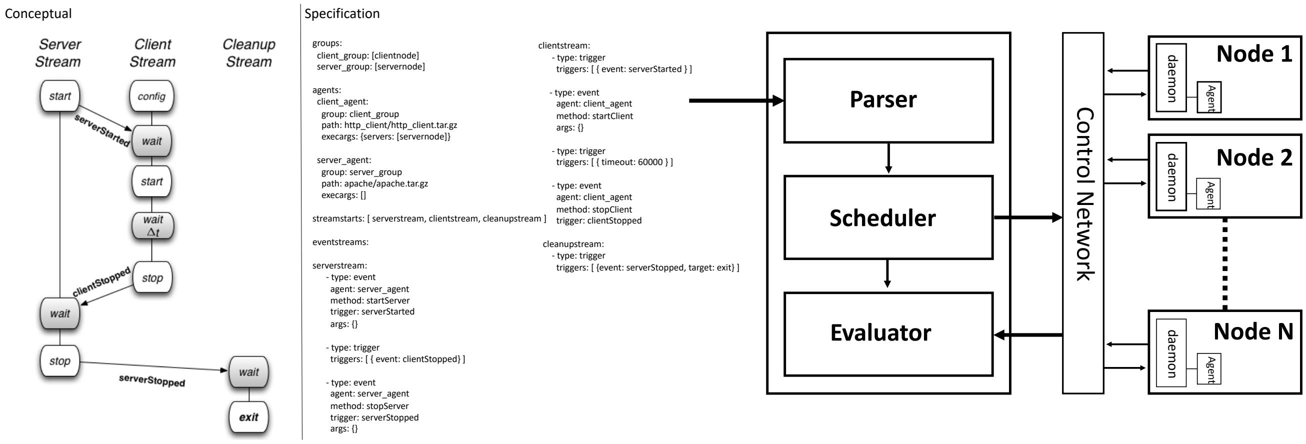
DETER Testbed Tools: The DETER testbed is an emulation-based testbed designed for cybersecurity experiments, especially those with malicious code [5, 39] . When the testbed launched in 2005, it had a GUI-based tool, SEER, that enabled the experimenter to visualize and monitor traffic on the experiment links and associate traffic generators with nodes at runtime via point-and-click mechansisms [34]. The DETER testbed provides an assortment of experiment management and topology construction tools [22]. More recently, it proposed a system to specify distributed workflows, which is currently under development [26].

GENI Testbed Tools: The Global Environment for Network Innovations (GENI) experimental facility is a federation of testbed resources contributed by various institutions including the testbeds listed above [24]. GENI is instrumented for the collection, analysis, and pooling of measurements from multiple locations and provides Fabric [19] and Ansible [3] to enable experimenters to customize and run experiments [18]. It also uses a LabWiki that allows experimenters to describe and instrument an experiment, execute it, and collect results [31]. GENI's OMF is a testbed control, measurement, and management framework that allows configuration and control for testbed-based experiments. OMF is event-driven and provides a set of trigger-based and time-based directives for experiment definition [24, 30]. The Orbit Experiment Description Language (ODEL) provides the OnEvent directive to synchronize and order processes. The OMF experiment controller comes with a default set of events that can be extended with the `defEvent` directive. OMF natively supports collection of measurement data.

Emerging Testbeds: The testbeds mentioned above are either transforming or retiring as new cloud-based and virtualization-based testbeds emerge. We mention a few here to complete our discussion on testbeds and tools. The FABRIC testbed, started in October 2019, is a unique national research infrastructure for exploratory research at-scale in networking, cybersecurity, distributed computing and storage systems, machine learning, and science applications. It's goal is to enable nationwide instrumentation with network elements equipped with large amounts of compute and storage resources, interconnected by high speed, dedicated optical links [4]. The Chameleon testbed, started in July 2015, provides a configurable experimental testbed for cloud research and education communities. It allows experiments to address the challenges of high-level cloud research, such as cloud scheduling, cloud platforms, and cloud applications, and low-level problems in hardware architecture, systems research, network configuration, and software design [9]. EdgeNet, started in 2018, is a modern distributed edge cloud, which incorporates advances in cloud technologies in order to allow experimenters to use standard Kubernetes tools and technologies to deploy an application across the EdgeNet infrastructure [16]. The DComp testbed, started in 2019, is a large-scale testbed, combining EVPN-based network isolation with customized nodes, commodity switches, and modular software to create flexible and adaptable strategies to provision network emulation and infrastructure services on a per-experiment basis [17].

3 Key Features

This section discusses the key features of the MAGI tool for instrumenting experiments with high-level directives to control concurrency and graceful handling of failure. An experiment typically starts with a conceptual model that is translated into a specification and orchestrated on a testbed, as shown in Figure 1. An experiment specification is a sequence of steps that describe an individual run of a distributed system with a specific set of parameters, configuration of background traffic



```
groups:
  client_group: [clientnode]
  server_group: [servernode]

agents:
  client_agent:
    group: client_group
    path: http_client/http_client.tar.gz
    execargs: {servers: [servernode]}

  server_agent:
    group: server_group
    path: apache/apache.tar.gz
    execargs: []

streamstarts: [ serverstream, clientstream, cleanupstream ]

eventstreams:

serverstream:
      - type: event
        agent: server_agent
        method: startServer
        trigger: serverStarted
        args: {}

      - type: trigger
        triggers: [ { event: clientStopped} ]

      - type: event
        agent: server_agent
        method: stopServer
        trigger: serverStopped
        args: {}

clientstream:
    - type: trigger
      triggers: [ { event: serverStarted } ]

   - type: event
     agent: client_agent
     method: startClient
     args: {}

    - type: trigger
      triggers: [ { timeout: 60000 } ]

    - type: event
      agent: client_agent
      method: stopClient
      trigger: clientStopped

 cleanupstream:
    - type: trigger
      triggers: [ {event: serverStopped, target: exit} ]
```

Figure 1: Experiment Orchestration: An experiment starts with a conceptual model that is encoded as a specification and orchestrated on the testbed

components, and mapping of components to testbed node and network elements. Experiment orchestration is the process of executing this sequence of steps. We discuss them in detail below.

3.1 Specification

The experiment specification is a concise description of the experiment's execution as a sequence of steps. Figure 1 outlines the process of experiment orchestration from conceptualization, through specification, and then execution on a testbed. We now illustrate this process with a simple client-server example. The steps at the conceptual level include: starting servers, starting clients, running the experiment for some period of time Δt, and then analyzing the results. These steps are translated into a specification with the following directives: (a) *Groups*: Nodes, the physical or virtual machines that form the topology, are organized into *groups* based on the role they play in the experiment, such as a *client* or a *server*. The *group* directive is the only coupling between the experiment specification and the experiment topology. As the experiment scales from a few nodes to hundreds of nodes, only this part of the experiment specification needs to be modified. (b) *Agent*: The agent code that implements the role behavior or functionality in the experiment. The specification also includes a mapping of the agents to the groups as seen in Figure 1. The `client_agent` is mapped to all nodes in the `client_group`. The scheduler deploys the agents implementation, located at the `path` on the nodes within the group, and parameterizes them as specified in `execargs`. (c) *Event*: Each event corresponds to a method implemented in the agent and is invoked by the scheduler. In this example, a client has a `startclient` method that requests a file every five seconds from the server, the size is randomly distributed between 1KB to 10KB. All events are non-blocking and run asynchronously. After the event is scheduled, the scheduler moves on to the next event in the stream. (d) *Eventstreams*: A collection of related events is organized into blocks that formulate the desired behavior in the experiment. They have a unique identifier that can be used to execute the event stream at the start of the experiment with `streamstarts` or after evaluating a trigger. (e) *Trigger*: An experiment stream can define time-based or condition-based barrier synchronization point, a *trigger*, where the scheduler must wait. When the condition is satisfied, the evaluator returns the result to the scheduler, and the event stream is unblocked. We discuss these in more detail below.

For statistically sound results, a series of identical experiments is typically executed. A series of different experiments is used to answer an experimental question, such as finding an optimal setting for a system parameter by varying the value of that parameter across experiments.

3.2 Orchestrator

The orchestrator is the heart of the MAGI tool and is invoked through the command-line. It consists of three main parts as shown in Figure 1, The *parser* reads the experiment script and identifies the `group`, `agents`, `events`, `eventstream` and `triggers` in the experiment. The decomposition of events into event streams provides the flexibility of scheduling them at multiple times in the experiment based on the results received from the experiment execution.

The *scheduler* starts a thread to handle each eventstream and sends events to all the nodes in the experiment. An event invokes an agent method on the group of nodes with the specified parameters. Events are nonblocking, hence once an event is sent out to the daemon, the scheduler moves onto the next event in the eventstream. Each event can define a return value that will be sent back to the evaluator on completion of the event. These return values are labeled to indicate which event generated them. In the client-server example, the first event in the serverstream, called startServer, labels the return value as serverStarted. Multiple event streams can be scheduled to run concurrently as the experiment executes.

The *evaluator* receives the return values from the nodes in the experiment and collates the responses. Every event in the experiment sends a response back to the orchestrator to enable first class logging and error handling. An experiment eventstream can define a time-based or a condition-based barrier synchronization point, called *triggers*. In time-based triggers, the scheduler waits for the specified amount of time, in milliseconds, before proceeding with the execution of the event stream. For example, [{timeout: 60000}] will cause the scheduler to wait for 60 seconds before processing the next event in the clientstream eventstream. In condition-based triggers, the scheduler must wait for the agents to return from the labeled event. The evaluator receives the return values from the agents and tries to satisfy the barrier condition. Barrier conditions are evaluated as a series of boolean return values from the agent that indicate success or failure of the event. Return values may also be a result string. For example, triggers:[{event: clientStopped}] in the serverstream eventstream, will cause the scheduler to wait until all the stopClient method on the remote nodes is executed by the agents. The daemon then returns the results labeled with clientStopped, which is evaluated by the evaluator.

By default, the evaluator waits for *all* agents within the agent group, but for graceful error handling in large topologies, experiments can specify a smaller *count* number of return values for success. When the condition is satisfied, the evaluator returns the result to the scheduler and the event stream is unblocked. Triggers can also be used to specify conditional branching in the eventstream based on the value of the returned result. We discuss an example in Section 4. The target directive within a trigger, as shown in the cleanupstream eventstream, enables an experiment to loop in the current eventstream or terminate the current eventstream to start another eventstream. The target directive is widely used to provide error handling, experiment analysis, and procedural complexity within the experiment.

3.3 Daemons and Agents

The lightweight daemon provides a conduit for control on the experiment nodes and runs on every node in the experiment. The daemon maintains the group memberships and is responsible for deploying agent implementations, receiving the event commands, invoking the agent methods based on the commands, and sending the results back to the evaluator. This is all done asynchronously. An agent runs in two modes: thread-based or process-based. In the threaded mode, the daemon runs the agent as a thread in its own process space, which simplifies status and failure handling. In the process mode, the daemon runs the agent in a process space separate from itself. If possible, the daemon communicates with the agent via a pipe or a socket for status and failure handling. Some agents are off-the-shelf components and have no mechanism to communicate with the daemon, limiting the ability of status and failure handling. One of the important goals of the daemon's task is to support error handling and transmit the errors back to the orchestrator when they occur. When evaluating large systems under stress and attack, the experiment may have many unexpected errors and failures due to interaction between different processes or incorrect assumptions about the concurrency involved. The MAGI tool generates detailed log traces with multiple levels of detail to understand how the execution evolved during the experiment. Logs can be saved locally as well as managed by a log collector and saved in a MongoDB database.

The MAGI tool is organized into two parts; core and agent modules. The core system consists of the orchestrator, daemon, messaging utilities, and log and data management infrastructure. It is implemented in 15K lines of Python code. The agent modules are agent function implementations and the code base consists for more than 20+ agents to generate web traffic, data sharing, microblogging, irc, voip, videostreaming, specific attack traffic agents as well as tcpdump, packetcounter, and file and process monitoring agents. This code base is constantly evolving, as users add more agents, and is currently at 35K lines of code.

MAGI has also been used in Emulab, GENI, and mininet environments. The MAGI toolkit has a modular structure that supports defining testbed-specific configuration, such as directory structures, topology and network interface information. When the tool is invoked on other testbeds, the testbed parameters can be specified in the configurations files.

4 Case Studies

This section illustrates how the MAGI tool enables novice students as well as sophisticated experimenters to to run complex experiments. We illustrate some of the mechanisms introduced in the previous section. All the experiments were conducted on a emulation-based testbed.

4.1 Senior-level Undergraduate Course

The MAGI tool has been used to develop and evaluate programming assignments for CS353, introduction to computer networks, a senior-level class on networking and distributed systems at USC. The course programming assignments required the students to progressively build a large networked and distributed system with several milestones and checks set up during the course to support the students. For example, the students were required to develop a text-based, multi-user chat client and server system over a period of twelve weeks. In the first few weeks, they developed the client application following the detailed protocol defined in the assignment and

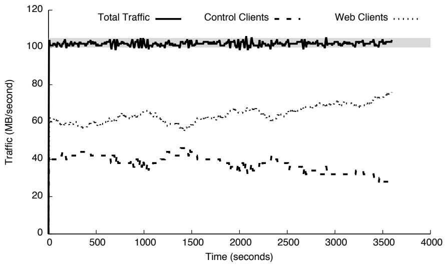


Figure 2: Closed-loop orchestration

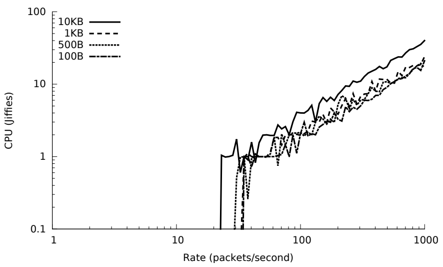


Figure 3: Impact of rate of event messages

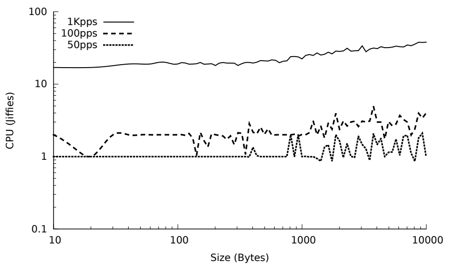


Figure 4: Impact of size of event messages

evaluated it against the instructor's server. Then they developed the server that supported one-on-one chatting between the clients. For the final part of the assignment, they added support for one-to-many chatting between the clients.

One of the challenges students face when developing such systems is self-assessing the correctness and performance of their system within a *real* distributed deployment beyond their personal laptop machine. As instructors, we want to facilitate mechanisms for the students to check the implementation of the server with different client implementations and with multiple clients at the same time over the network. However, as a student, such self-assessment is extremely hard, being a novice in an introductory course on distributed systems.

To facilitate self-assessment, we developed a pair of client and server agent modules that deployed their implementation as a process-based agent on multiple systems. During the discussion sessions for the course, we conducted hands-on tutorials with example testcase inputs to ensure the students learned how to evaluate their implementations at each milestone. During the first part of the assignment, the students could rapidly self-assess their client implementation with the instructor's server. During the second part, the MAGI tool randomly chose another classmate's client implementation (already verified for correctness from the first part) to self-assess their server implementation. During the final part of the assignment, we allowed the students to self-assess by deploying up to thirty clients with their server implementation. The MAGI tool proved invaluable in allowing the students to develop and evaluate their solutions in a consistent manner, get immediate feedback on the correctness of the protocol implementation, and truly explore development in a distributed setting.

We used the same setup to the grade the assignments. The continuity of the environment from the assignment development and self-assessment phase to the assignment grading phase made grading straightforward. Students who did not adhere to the protocol exactly, or could not complete the full protocol implementation due to their workload, required some manual deployment and grading. The MAGI tool has been used for four years with class sizes varying from 40 to 75 students.

4.2 Feedback Loops

Some experimentation scenarios require different teams to interact, while limiting the access to parts of the experimentation environment. Typical examples include attack-defense games [11, 27] and capture-the-flag games [12]. Each team attempts to achieve their goals in the presence of benign or adversarial disturbances. In this example, we illustrate how the MAGI tool can be used to facilitate such experiments. While the orchestration of a full game is complex and beyond the scope of this paper, we demonstrate how the MAGI tool can be used to create a closed-feedback loop for experimentation. We believe creating such closed-feedback loops is a fundamental building block for automated development of adversarial games.

In this illustrative game, the goal is to maintain the amount of traffic on a link within the range of 100MBps–105MBps. Our experiment topology consists of 3000 web clients served by a web-server farm of 50 apache2 servers connected in a canonical dumbbell topology. The web clients are partitioned into two groups: 2000 web clients that periodically request a random-sized file from a server and 1000 control clients that change the size of their request to ensure the traffic goal is achieved with a *sensor, compute and actuate* agent. A *sensor* agent measures the amount of traffic on the link; a *compute* agent devises a control action to increase, decrease, or maintain the same traffic; and an *actuate* agent executes the control action. With condition-based triggers, the experiment specification can branch to different eventstreams to modulate amount of traffic on the link as shown in the specification sample below:

```
- type: trigger
   triggers: [ { event: eth0Sensed, result: '-1', target: 'increaseTraffic' },
   { event: eth0Sensed, result: '1', target: 'reduceTraffic' },
   { event: eth0Sensed, result: '0', target: 'controlLoop'}]
```

This trigger waits for the sensor results to return periodically to the evaluator. The sensed value is compared to the target range of 100MBps–105MBps to compute an actuation action to increase traffic, reduce traffic, or just loop back to the `controlLoop`.

The resulting graph in Figure 2 shows that the controlled clients adjust their load as required to successfully maintain the traffic on the link within the specified threshold. The x-axis shows experiment time and the y-axis shows the volume of traffic on the link in MBps. The gray box indicates the threshold of 5MBps of tolerated fluctuation on the link. The web clients in the graph are exogenous to this experiment and only the control clients can be orchestrated in this experiment.

4.3 Integrated System Development and Evaluation

This example illustrates how five teams in the DARPA SAFER research project [13] used the MAGI tool to develop and evaluate adversary-resistant communication technology to circumvent censorship in Tor. The experiments were complex, as they required thousands of users communicating with applications such as instant messaging, electronic mail, social networking, streaming video, voice over Internet protocol (VoIP), and video conferencing, while attackers were attempting to discover the identity and location of the users and block the communication. The evaluation involved conducting a range of attacks to check the effectiveness of a team's solution across several vectors, such as user anonymity, message integrity and delivery, protocol inspection, client connectivity, and the ability of the solution to ensure speedy delivery.

The experiments needed to support the configuration and deployment of Tor, a large complex distributed system, along with the orchestration of the technology developed by each team on a wide range of topological representations and traffic environments. As each prototype technology matured, the test and evaluation performer had to support the orchestration of multiple teams, which required integrating their systems both for demonstrations and for system analysis. The scale of their experiments ranged from tens to hundreds of physical nodes on the DETER testbed.

Each team evaluated their technology with wide-area microblogging and data sharing scenarios [11, 15, 36, 38]. The underlying Tor network consisted of two evaluation topologies, one with 32 servers with 10 client machines per server, and 24 servers with 12 client machines per server, for a total of 320 and 288 client machines respectively. To simulate a larger number of client participants, some experiments launched 16 client process modules on each client machine, creating 5120 client processes that were managed by the MAGI tool. The teams developed a Tor agent that bootstrapped all the Tor-related directories, relays, bridges and clients with a single command. Once the setup was completed, the agent signaled that the Tor system was ready. Each team then launched their respective agent modules that implemented the functionality of their solution. After the technology is deployed, the background and attack traffic agents generate the evaluation traffic and start monitoring the experiment. The team created more than 390 unique experiments on the DETER testbed and each experiment was run using the MAGI tool hundreds of times for statistically sound results.

The project also conducted larger red teaming exercise with a three-tier node "core-stub-node" topology with 45 client nodes generating web and VoIP traffic [11]. Each client had a microblogging, VoIP, and data sharing traffic module. The microblogging module generated curl requests to fetch files of sizes less than 500K from a remote web server, whereas the VoIP module generated fixed bit-rate traffic between the client and a destination server for a randomly selected period of time. Additionally, the teams conducted periodic integrated demonstrations for the research sponsor. The culminating demonstration at the end of the project was conducted with all five technologies deployed over a hundred client topology. This demonstration experiment had over 30 different agent modules that generated microblogging, data sharing, and VoIP traffic in addition to the anti-censorship defense technology modules developed by each team.

4.4 Cyber-physical Systems

The modern power grid, with thousands of digital sensors monitoring the conventional and renewable power sources, energy storage systems, and smart loads, creates a large and complex distributed cyber-physical system [8]. This system poses novel challenges in capturing the vast amounts of sensor data and developing command and control algorithms for smart cities and smart homes.

We discuss how the MAGI tool facilitated the evaluation of a real-time distributed optimization algorithm for monitoring of power flow oscillation patterns in large power system networks. The solution evaluated two variants of the control algorithm: a centralized algorithm and a distributed consensus-based estimation algorithm in the presence of DoS attacks.

The experiment was run with an 18 node network topology overlaid on a IEEE 39 bus power system. Both control algorithms were developed in C and deployed using process agent modules in the experiment. In addition to the estimation control agents, the experiment also generated background web traffic and launched high volume DoS attacks on the communication links between the areas to evaluate resiliency. The results are not included in the paper due to space constraints [40].

5 Performance

To evaluate the performance of MAGI, we set up a two-node experiment and systematically vary the volume of control messages from the scheduler to the sink node. Both nodes are quad-core Xeon 3.0GHz processor with 2 GB memory.

On the scheduler node, we deploy a stress agent that can generate messages of the specified size and rate and send them to the sink node. We measure CPU utilization every second in jiffies for all child processes and threads within the system by tracking `/proc/stat` and `/proc/<processid>/stat` files. We get the jiffies spent in both user mode and kernel mode and compute CPU utilization using standard techniques [23]. We show CPU utilization in jiffies, rather than in CPU utilization percentage, for better resolution in the figures below.

We vary the control message rate from 1 packet/second to 1000 packets/second in steps of 1 packet/second. The size of each message is constant at 1000B and each evaluation round lasts for 1000 seconds. Figure 3 shows the CPU utilization in jiffies on the y-axis in log scale with varying message rate on the x-axis in log scale. At packet rates below 10 packets/sec, the CPU utilization is at or below 1 jiffy and hence not shown. The CPU utilization is directly proportional to the rate of messages. This implies that the processing overhead is uniform at the sink node even as the rate increases. The maximum CPU utilization at packet size of 10KB is 25 jiffies or 6%CPU at peak load. This result shows that our system scales well with an increase in the rate of messages and that the performance characteristics degrade gracefully with increased load.

We vary the message payload size from 0 bytes to 10K bytes in steps of 10B while keeping the message rate constant at 100 packets/sec. Each evaluation round lasts for 1000 seconds. Figure 4 shows the CPU utilization in jiffies on the y-axis in log scale with varying message size on the x-axis in log scale. At packet rates below 50 packets/sec, the CPU utilization is at or below 1 jiffy and hence not shown. The CPU utilization remains almost constant with the change in message size. The marginal increase in CPU utilization is due to the increased amount of data being processed-in from the socket-level transport interface and being stored and passed to the system. The maximum CPU utilization at the rate of 100 packets/sec is 4 jiffies or 1% CPU and at rate of 1000 packets/sec is 38 jiffies or 9% CPU when the packet size is 10KB.

The above control message volume is well below what is typical generated in experiments. For example, in the case studies discussed in Section 4, we generated a maximum of 40KBps when experimenting with 10K agents at a maximum rate of 200Bps.

6 Retrospective Takeaways

This effort offers several important lessons. The MAGI tool made the process of running experiments substantially easier. First, it is *topology agnostic*. The experiment, defined as a sequence of steps, is coupled to the topology using only the `groups` directive in the specification. A change in the topology structure or scale results in changes in the group specification. Since no other changes are required, experimenters widely used this feature to develop and test the experiment at small scales before deploying large scale experiments. Second, it allows the experimenter to exploit concurrency where ever possible in the experiment design. For example, it allows the experimenter to rapidly and concurrently deploy and start a range of traffic generators that are not dependent on each other using events and event streams. When the order of unrelated events is not relevant, the experimenter can leave the order unspecified, providing greater flexibility and experiment execution speed. When the order of the events is important, for example, starting clients only after all the servers have started, a `trigger` can be used to order the events. Third, it has multiple levels of concurrency: multiple event streams can be scheduled at the orchestrator; multiple agents can be deployed and managed by the daemon on the local node; and the experiment itself is distributed across multiple nodes. Fourth, the tool combines logging, events, and triggers to provide mechanisms for specifying, notifying, and handling failures critical for experiment reliability. The authors believe the above features were key to MAGI's success as they enabled developing adaptive experimentation environments that allowed rigorous evaluations of the systems under test.

It is impossible to anticipate all possible components required by future experiments. Therefore, MAGI provides a systematic way for researchers to add new components, such as new attack tools, to the framework as agent modules. Our experience is that a researcher will choose to use MAGI because of its wide range of supported modules, and will add their own novel modules to MAGI because they find it convenient and productive to do so. This has lead to a virtuous cycle in which the growing set of community developed modules and tools attracts even more users to using MAGI.

We are starting to develop the next generation orchestration tool for the Merge testbed platform [25]. In addition to the above listed features, one of the goals for the next generation orchestration tool will be to ensure the footprint of the daemon is minimal so that it can be effectively used in highly virtualized environments.

7 Conclusion

This paper is a retrospective on the development and use of an experiment orchestration tool. The MAGI tool enabled the development and evaluation of large and complex networked and distributed systems in testbed environments through rapid experimentation in representative scenarios. We described the key components of its architecture and presented several illustrative examples of its use in a variety of settings ranging from undergraduate education, anonymous communication, cyber-physical systems, and attacker-defender games. We discussed the key takeaways from the design, including topology-agnostic specification along with high-level directives for concurrency, message passing, and error handling, that we believe facilitated experimentation. We hope these insights and takeaways, distilled with the usage of our MAGI tool over a period of seven years, provide guiding principles that will aid in developing the next-generation experiment management tools.

Availability: The MAGI documentation and sample experiments are located at https://montage.deterlab.net/magi/. The MAGI code base is installed on the DETER testbed and available at https://github.com/deter-project/magi.

References

[1] Jeannie Albrecht, Christopher Tuttle, Alex C Snoeren, and Amin Vahdat. Planetlab application management using Plush. *ACM SIGOPS Operating Systems Review*, 40(1):33–40, 2006.

[2] Jeannie R Albrecht, Ryan Braud, Darren Dao, Nikolay Topilski, Christopher Tuttle, Alex C Snoeren, and Amin Vahdat. Remote control: Distributed application configuration, management, and visualization with Plush. In *LISA*, volume 7, pages 1–19, 2007.

[3] Ansible: Simple, agentless IT automation. https://https://www.ansible.com/.

[4] Ilya Baldin, Anita Nikolich, James Griffioen, Indermohan Monga, Kuang-Ching Wang, Tom Lehman, and Paul Ruth. Fabric: A national-scale programmable experimental network infrastructure. *IEEE Internet Computing*, 23(6):38–47, 2019.

[5] Terry Benzel. The science of cyber security experimentation: the DETER project. In *Proceedings of the 27th Annual Computer Security Applications Conference*, pages 137–148, 2011.

[6] Mark Berman, Jeffrey S Chase, Lawrence Landweber, Akihiro Nakao, Max Ott, Dipankar Raychaudhuri, Robert Ricci, and Ivan Seskar. GENI: A federated testbed for innovative network experiments. *Computer Networks*, 61:5–23, 2014.

[7] Tomasz Buchert, Cristian Ruiz, Lucas Nussbaum, and Olivier Richard. A survey of general-purpose experiment management tools for distributed systems. *Future Generation Computer Systems*, 45:1 – 12, 2015.

[8] Aranya Chakrabortty and Pramod Khargonekar. An introduction to wide-area control of power systems. In *American Control Conference*, pages 6758–6770, 2013.

[9] The chameleon testbed: A configurable experimental environment for large-scale cloud research. https://chameleoncloud.org.

[10] Chef: Helping IT operators achieve more. https://www.chef.io/.

[11] Sandy Clark, Chris Wacek, Matt Blaze, Boon Thau Loo, Micah Sherr, Clay Shields, and Jonathan Smith. Collaborative red teaming for anonymity system evaluation. In *Workshop on Cyber Security Experimentation and Test*, 2012.

[12] Crispin Cowan, Seth Arnold, Steve Beattie, Chris Wright, and John Viega. Defcon capture the flag: Defending vulnerable code from intense attack. In *Proceedings DARPA Information Survivability Conference and Exposition*, volume 1, pages 120–129. IEEE, 2003.

[13] DARPA Open Catalog: Publications from the SAFER Program. https://www.darpa.mil/opencatalog?ocSearch=safer&sort=program&ocFilter=publication.

[14] Eric Eide and Leigh Stoller. An experimentation workbench for replayable networking research. In *4th USENIX Symposium on Networked Systems Design & Implementation*, Cambridge, MA, 2007.

[15] David Fifield, Nate Hardison, Jonathan Ellithorpe, Emily Stark, Dan Boneh, Roger Dingledine, and Phil Porras. Evading censorship with browser-based proxies. In *Privacy Enhancing Technologies*, pages 239–258, Berlin, Heidelberg, 2012.

[16] Timur Friedman, Rick McGeer, Berat Can Senel, Matt Hemmings, and Glenn Ricart. The EdgeNet system. In *IEEE 27th International Conference on Network Protocols*, 2019.

[17] Ryan Goodfellow, Stephen Schwab, Erik Kline, Lincoln Thurlow, and Geoff Lawler. The DComp testbed. In *Workshop on Cyber Security Experimentation and Test*, Santa Clara, CA, 2019.

[18] James Griffioen, Zongming Fei, Hussamuddin Nasir, Charles Carpenter, Jeremy Reed, Xiongqi Wu, and Sergio Rivera. The GENI desktop. In *The GENI Book*, pages 381–406. Springer, 2016.

[19] Adrian Hannah. Fabric: a system administrator's best friend. *Linux Journal*, 2013(226):3, 2013.

[20] Fabien Hermenier and Robert Ricci. How to build a better testbed: Lessons from a decade of network experiments on emulab. In *International Conference on Testbeds and Research Infrastructures*, pages 287–304. Springer, 2012.

[21] Alefiya Hussain. MAGI: Montage AGent Infrastructure. https://montage.deterlab.net/magi/.

[22] Alefiya Hussain and Jennifer Chen. Montage Topology Manager: Tools for Constructing and Sharing Representative Internet Topologies. *Technical Report-684*, 2012, 2012.

[23] Kernel timer systems. https://elinux.org/Kernel_Timer_Systems.

[24] Rick McGeer, Mark Berman, Chip Elliott, and Robert Ricci. *The GENI book*. Springer, 2016.

[25] MERGE: A Testbed platform. http://mergetb.org.

[26] Jelena Mirkovic, Genevieve Bartlett, and Jim Blythe. DEW: Distributed experiment workflows. In *Workshop on Cyber Security Experimentation and Test*, 2018.

[27] Jelena Mirkovic, Peter Reiher, Christos Papadopoulos, Alefiya Hussain, Marla Shepard, Michael Berg, and Robert Jung. Testing a collaborative DDoS defense in a red team/blue team exercise. *IEEE Transactions on Computers*, 57(8):1098–1112, 2008.

[28] Isaac Pedisich. shremote: Execute commands remotely over SSH. https://github.com/isaac-ped/Shremote.

[29] Larry Peterson. PlanetLab: Its been a fun ride. https://www.systemsapproach.org/blog/its-been-a-fun-ride.

[30] Thierry Rakotoarivelo, Maximilian Ott, Guillaume Jourjon, and Ivan Seskar. OMF: a control and management framework for networking testbeds. *ACM SIGOPS Operating Systems Review*, 43(4):54–59, 2010.

[31] Niky Riga, Sarah Edwards, and Vicraj Thomas. The experimenter's view of GENI. In *The GENI Book*, pages 349–379. Springer, 2016.

[32] Haakon Ringberg, Matthew Roughan, and Jennifer Rexford. The need for simulation in evaluating anomaly detectors. *ACM SIGCOMM Computer Communication Review*, 38(1):55–59, 2008.

[33] Saltstack: Control and secure your infrastructure. https://saltstack.com.

[34] Stephen Schwab, Brett Wilson, Calvin Ko, and Alefiya Hussain. SEER: A security experimentation environment for DETER. In *Workshop on Cyber Security Experimentation and Test*, 2007.

[35] Terraform: Use infrastructure as code. https://https://www.terraform.io/.

[36] Zachary Weinberg, Jeffrey Wang, Vinod Yegneswaran, Linda Briesemeister, Steven Cheung, Frank Wang, and Dan Boneh. Stegotorus: a camouflage proxy for the tor anonymity system. In *Proceedings of the ACM conference on Computer and communications security*, pages 109–120, 2012.

[37] Brian White, Jay Lepreau, Leigh Stoller, Robert Ricci, Shashi Guruprasad, Mac Newbold, Mike Hibler, Chad Barb, and Abhijeet Joglekar. An integrated experimental environment for distributed systems and networks. *SIGOPS Oper. Syst. Rev.*, 36(SI):255–270, December 2003.

[38] David Isaac Wolinsky, Henry Corrigan-Gibbs, Bryan Ford, and Aaron Johnson. Dissent in numbers: Making strong anonymity scale. In *Proceedings of the Conference on Operating Systems Design and Implementation*, page 179–192, USA, 2012.

[39] John Wroclawski, Terry Benzel, Jim Blythe, Ted Faber, Alefiya Hussain, Jelena Mirkovic, and Stephen Schwab. DETERLab and the DETER Project. In *The GENI Book*, pages 35–62. Springer, 2016.

[40] Jianhua Zhang, Prateek Jaipuria, Aranya Chakrabortty, and Alefiya Hussain. A distributed optimization algorithm for attack-resilient wide-area monitoring of power systems: Theoretical and experimental methods. In *International Conference on Decision and Game Theory for Security*, pages 350–359. Springer, 2014.