Dapper

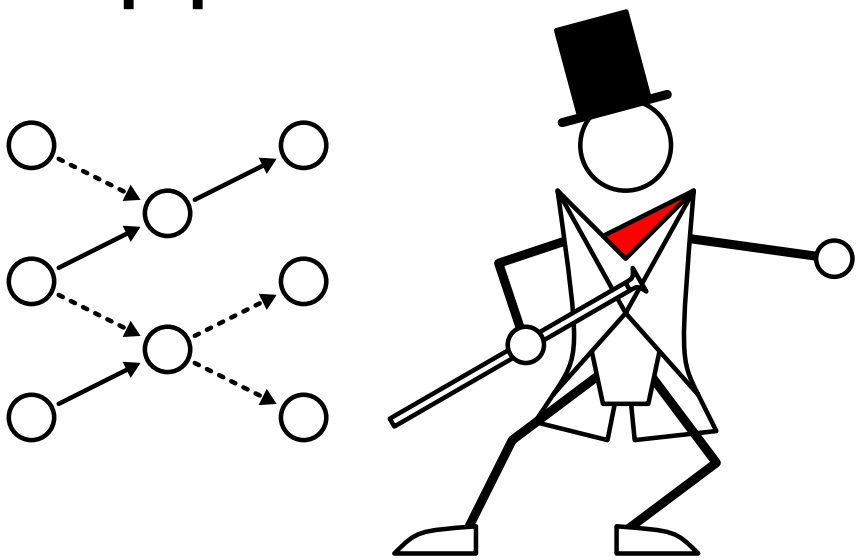
The Distributed and Parallel Program Execution Runtime

Roy Liu

December 2010

Contents

Dapper



Chapter 1

Introduction

1.1 Why Dapper?

We live in interesting times, where breakthroughs in the sciences increasingly depend on the growing availability and abundance of commoditized, networked computational resources. With the help of the cloud or grid, computations that would otherwise run for days on a single desktop machine now have distributed and/or parallel formulations that can churn through, in a matter of hours, input sets ten times as large on a hundred machines. As alluring as the idea of strength in numbers may be, having just physical hardware is not enough – a programmer has to craft the actual computation that will run on it. Consequently, the high value placed on human effort and creativity necessitates a programming environment that enables, and even encourages, succinct expression of distributed computations, and yet at the same time does not sacrifice generality.

Dapper, standing for **D**istributed **a**nd **P**arallel **P**rogram **E**xecution **R**untime, is one such tool for bridging the scientist/programmer's high level specifications that capture the essence of a program, with the low level mechanisms that reflect the unsavory realities of distributed and parallel computing. Under its dataflow-oriented approach, Dapper enables users to code locally in Java and execute globally on the cloud or grid. The user first writes codelets, or small snippets of code that perform simple tasks and do not, in themselves, constitute a complete program. Afterwards, he or she specifies how those codelets, seen as vertices in the dataflow, transmit data to each other via edge relations. The resulting directed acyclic dataflow graph is a complete program interpretable by the Dapper server, which, upon being contacted by long-lived worker clients, can coordinate a distributed execution.

Under the Dapper model, the user no longer needs to worry about traditionally ad-hoc aspects of managing the cloud or grid, which include handling data interconnects and dependencies, recovering from errors, distributing code, and starting jobs. It provides an entire Java-based toolchain and runtime for framing nearly all coarse-grained distributed computations in a consistent format that allows for rapid deployment and easy conveyance to other researchers. The words of the financier and statesman Bernard Baruch, however, best sum up Dapper's larger purpose:

> If all you have is a hammer, everything looks like a nail.

Thus, as much as it is a computer system, Dapper espouses a broadly applicable way of thinking about dataflow-driven distributed computing.

To offer prospective users a glimpse of the system's capabilities, we quickly summarize many of Dapper's features that improve upon existing systems, or are new altogether:

- A code distribution system that allows the Dapper server to transmit requisite program code over the network and have clients dynamically load it. A consequence of this is that, barring external executables, updates to Dapper programs need only happen on the server-side.
- A powerful subflow embedding method for dynamically modifying the dataflow graph at runtime.
- A runtime in vanilla Java, a language that many are no doubt familiar with. Aside from the requirement of a recent JVM and optionally Graphviz Dot, Dapper is self-contained.
- A robust control protocol. The Dapper server expects any number of clients to fail, at any time, and has customizable re-execution and timeout policies to cope. Consequently, one can start and stop (long-lived) clients without fear of putting the entire system into an inconsistent state.
- Flexible semantics that allow data transfers via files or TCP streams.
- Interoperability with firewalls. Since your local cloud or grid probably sits behind a firewall, we have devised special semantics for streaming data transfers.
- Liberal licensing terms. Dapper is released under the New BSD License to prevent contamination of your codebase.
- Operation as an embedded application in environments like Apache Tomcat.
- Operation as a standalone user interface described in Chapter 2. With it, one can run off-the-shelf demos and learn core concepts from visual examples. By following a minimal set of conventions, one can then bundle one's own Dapper programs as execution archives, and then get realtime dataflow status and debugging feedback.

1.2 Related Work

1.2.1 Current and Previous Approaches

The distributed computing literature is immense; even the small part with a dataflow-oriented flavor is sizable. Should Dapper not meet the user's requirements and specifications, quite a few alternatives exist.

MapReduce [1] and Hadoop [2, 3] – Google's internal computing engine and Apache's open source rendition of it. While well-suited for Google-scale operations like counting word occurrences among billions of documents, map-reduce architectures, as implied by the name, are limited in topology. For multistage bioinformatics computations,

many otherwise taken for granted properties, like automatic handling of data interconnects, no longer hold. To be fair, however, systems like Hadoop have added benefits that Dapper leaves to the programmer, like a distributed data storage model.

Dryad [4] – Microsoft's .NET-based answer to Google infrastructure, and substantively the closest system to Dapper. Like Dapper, Dryad represents distributed dataflows as directed acyclic graphs, and even has the ability to dynamically modify said graph at runtime. The similarities end at overarching ideas, however; Dryad is written with Microsoft's proprietary .NET infrastructure, and is optimized for relational query pipelines. Moreover, the authors have not yet made Dryad publicly available. Until they do so, one misses out on key design details, such as those that went into runtime graph modification, and one cannot assess the system's usability.

Taverna [5] and other workflow systems – In case you're a bioinformatician looking for something that takes advantage of existing biological data processing pipelines, and not building your own. Scientific workflow systems tend to focus on end users with minimal programming experience, let alone the patience for it. They do not address infrastructural requirements, however, and leave the actual implementation of a workflow service to the programmer. Consequently, we see Dapper's role as an intermediate layer between the cloud or grid and workflow enactment systems.

1.2.2 Rationale and Background

In an unpublished technical report [6], Carnegie Mellon School of Computer Science Dean Randy Bryant argues that large-scale computing systems pioneered at Google/Yahoo/Microsoft's data centers pave the way for a new kind of supercomputing research dubbed **D**ata-**I**ntensive **S**uper**c**omputing (**DISC**). He mentions many types of scientific calculations with distributed formulations that would benefit from a massive scale-up. Although we initially conceived of and designed Dapper without the above formalities in mind, the fact that we did so out of need, along with the emergence of Dryad, MapReduce, Hadoop, and Taverna, all corroborate Bryant's views of the changing computing landscape. In light of DISC, we hope that Dapper, too, helps scientific pipelines achieve ever greater throughput while reducing programmer effort.

Despite the promise the future holds, something should be said about pioneering work which provides an intellectual foundation for all distributed, dataflow-oriented systems. We cite Paralex [7] in particular as perhaps the first concept system of how modular units of computation might execute on networked computer clusters, and pass data among each other to calculate a final result. The publication of Paralex nearly one and a half decades ago should not be cause for one to dismiss it; relatively speaking, the system was radical in a time when the increasing speed of uniprocessors subsumed the need for writing distributed programs. Now that processor speeds seem to have peaked and computer hardware is undergoing commoditization, the ideas behind Paralex should start to gain relevance.

1.3 Obtaining and Installing

Here's how to obtain Dapper and/or learn more about it:

- Downloads of source and Jar distributions from Google Code.
- Javadocs of member classes.
- A Git repository of browseable code.

To get up and running quickly, download the two Jars dapper.jar and dapper_ex.jar (modulo some version number x.xx). You will need to have Graphviz Dot and Java 1.6.∗+ handy. Start the user interface with the command

```
java -jar dapper.jar,
```

or, if your operating system associates the .jar extension with a JRE, by clicking on the icon. Drag and drop the Jar of examples, dapper.jar, into the box containing the "Archives" tree. You will see a few selections; select the one that says "ex.SimpleTest", and then press the "run" button. Now start at least four worker clients by repeatedly issuing the command

```
java -cp dapper.jar org.dapper.client.ClientDriver.
```

By now, you should see the user interface begin to step through the "Simple Test" computation. Although everything is happening on the local machine, the Dapper server embedded in the user interface is completely agnostic to the actual disposition of clients. Thus, fully distributed operation is intrinsically no harder than the steps laid out above.

Downloading and compiling the dapper-src.tgz source distribution is also possible. Upon unpacking, changing into the distribution base directory, and typing Make, you will find that build process attempts to use Apache Ivy to resolve external library dependencies. Although most are offered from the highly visible public Maven repository, one major dependency, the Shared Scientific Toolbox (SST), does not have the same availability. In the unlikely event of the SST web repository being unavailable or broken, please go here, download the sources, compile, and then type make publish. This will publish the SST to an Ivy repository residing in a local folder. When changing back into the Dapper directory, the user may then run test.py to verify basic Dapper functionality. Alternatively, Windows users may run the precompiled executable buildandtest.exe.

1.4 Future Directions and Contributing

We welcome you to join us in extending Dapper's reach. At the most basic level, consider spreading the word to your friends and colleagues, or linking to http://carsomyr.github.io/dapper/. You may also cite this user manual as a BibTeX entry:

```
@booklet{liu10,
    author  = {Roy Liu},
    title   = {Dapper: The Distributed and Parallel Program Execution  \
       Runtime},
    year    = {2010}
}
```

If you would like contribute in a direct way, however, do not hesitate to contact this author – any help with software development, system design, and/or documentation would be greatly appreciated. Below are just some future avenues of work:

- A specification language for laying out dataflow graph topologies. Such a language would support recursive constructs that elegantly capture the concept of subflow embedding.
- Stop-and-go persistence of dataflows to a database through Hibernate.
- While core functionality is already in place, Dapper lacks integration with current well-known deployment environments like Eclipse and JSP. Providing interfaces to these would definitely increase the system's popularity.

Chapter 2

Examples

By narrating the reader through multiple scenarios and example dataflows, we intend to teach Dapper's basic concepts and operation via the user interface. Moreover, the current chapter serves as a gentle introduction to basic Dapper operations and a bridge to the very technical discussion that will follow in Chapter 3.

2.1 Dapper Terminology

Before describing examples and their inner workings, let alone introducing the programing API of Chapter 3, we define some oft-used terminology. We put effort into choosing those that have resonance in other, analogous contexts. The list below, although by no means comprehensive, should serve as an cheat sheet of sorts:

- Server – A process managing the centralized command data structures and orchestrating the work of many machines.
- Client – A worker that reports to the Dapper server and runs a dataflow computation at the behest by the server. Upon completion, the client sits idle and awaits further instructions.
- Node – Represents a computation in the dataflow graph.
- Edge – Represents a data transfer between computations in the dataflow graph.
- Codelet – A basic, atomic, user-defined unit of computation executing on a Dapper client with a logical node binding; essentially a callback with arguments that consist of lists of input and output resources as well as user-defined parameters.
- Execution Archive – A special formulation of Dapper programs as Jars for inspection and loading by the Dapper user interface.

Dapper inverts the traditional cloud/grid computing system architecture; whereas usually the master machine sends commands to workers through some sort of cloud/grid frontend (or even the ssh program), Dapper clients report to their designated server, which has no idea of the computational resources available to it in advance. Upon determining eligibility of clients for the tasks at hand, the Dapper server then distributes code and otherwise configures clients for execution. The resulting system is a de facto distributed virtual machine environment where one need not worry about the bindings of computations with the physical machines that execute them. Consequently, Figure 2.1 illustrates the idea that the Dapper server bridges abstract dataflow specifications and physical machines.

2.2 Navigating the User Interface

To demonstrate the capabilities of the Dapper, we have built a lightweight, complementary user interface. The user interface fulfills three purposes – first, to impress upon users that dataflows have very real, highly visual manifestations;

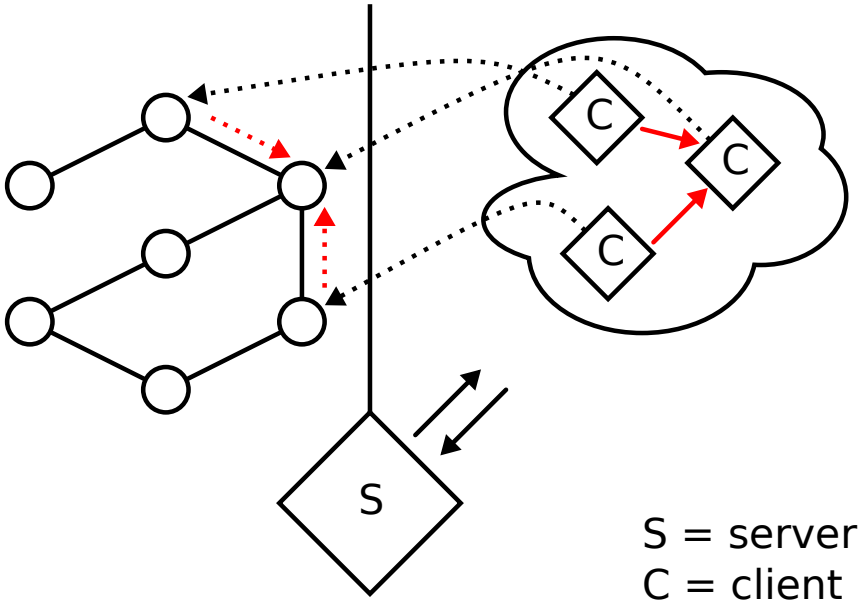


Figure 2.1: *The Dapper system diagram.*

second, to provide a minimalist platform for managing dataflows; and third, to exemplify operation of the Dapper server as a potentially useful embedded application.

We have designed the user interface to be as intuitive as possible; it is, quite literally, drag-and-drop by following the procedures in Section 1.3. Figure 2.2 lists components and their functions:

1. **Current Tab** – Contains the current dataflow being rendered.
2. **Execution Archive Tree** – A hierarchical view of execution archives. Upon receiving an execution archive from a drag-and-drop on this component, the user interface will inspect it for executable program code. For a more involved discussion, see Section 3.3.
3. **Run Selected Dataflow** – Runs the selected entry in the Execution Archive Tree.
4. **Unload Execution Archive** – Unloads the selected Execution Archive from the Execution Archive Tree.
5. **Special Options** – Enables/disables certain special options that subtly affect behavior:

    - Remove Finished [ALT + R] – Regulates whether or not to close tabs associated with successfully completed dataflows.
    - Take Screenshot (SVG) [ALT + S] – Saves a screenshot of the currently rendering dataflow in SVG format.
    - Take Screenshot (PNG) [ALT + P] – Saves a screenshot of the entire user interface in PNG format.
    - Purge Current [ALT + C] – Purges the current dataflow.

Upon selecting a tab, its associated dataflow will start rendering from repeated invocations of the Graphviz Dot graph drawing engine. Fundamentally, the representation of dataflows consist of nodes standing in for computations and (directed) edges standing in for data transfers; however, the color and outline of each node and edge carries meaning. Figure 2.3 contains an index of visual components along with their interpretations:

1. An idle node.

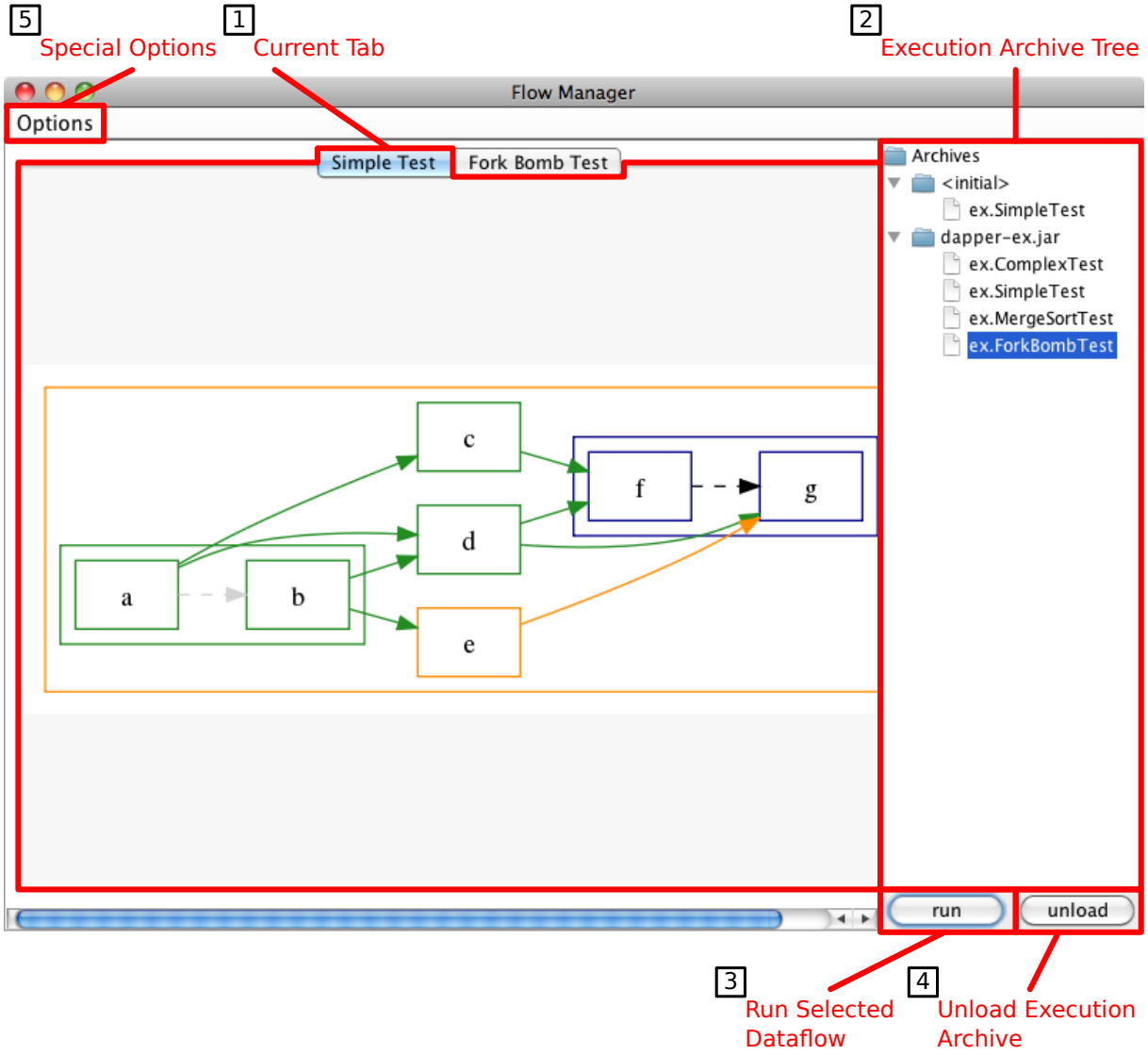


Figure 2.2: *The Dapper user interface.*

2. A node preparing for execution.
3. A finished node.
4. An executing node.
5. A node that has encountered an error, and remains to be started anew.
6. A node that embeds a subflow (see Section 3.2.2).
7. A HandleEdge (see Section 3.1.5).

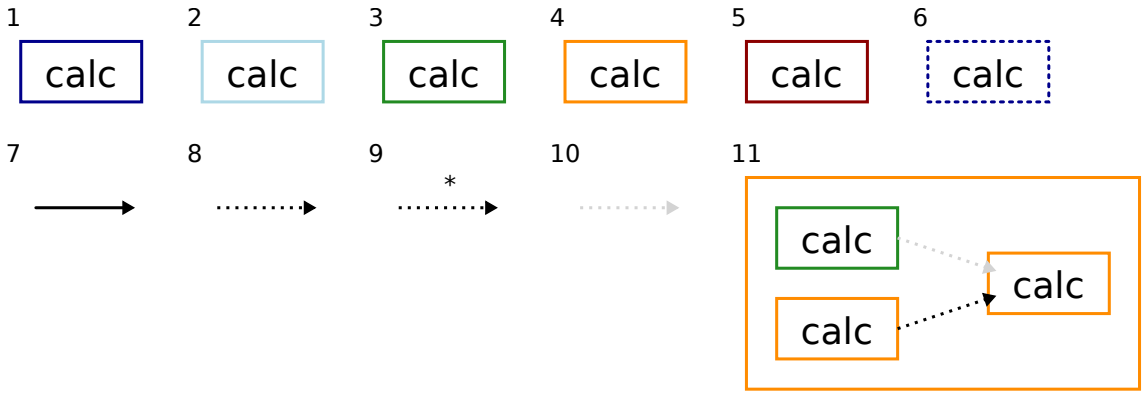


Figure 2.3: *A visual index of Dot-rendered components.*

8. A StreamEdge (see Section 3.1.5).
9. A StreamEdge with an inverted "connects to" relation.
10. A StreamEdge whose start node has finished.
11. A connected component induced by StreamEdges, where all member computations start simultaneously. While some members may finish before others, the Dapper server considers the whole equivalence class finished if and only if all members have finished (see Section 3.1.5).

2.3 Demonstrations

2.3.1 Simple Test

The Simple Test is a sampler of Dapper's functionality. One can run it with the provided test.py script or buildandtest.exe executable. Under the covers, a FlowManager user interface with an embedded Dapper Server, along with four Dapper Clients, are started. The dataflow shown in 2.4 steps through dummy calculations that do nothing but wait a random amount of time and finish. Although no data transfers happen, the simultaneous execution of weakly connected components induced over StreamEdges (delimited by a box around multiple nodes) demonstrates Dapper's ability to automatically infer dataflow execution semantics.

2.3.2 Complex Test

Following on the heels of Simple Test, the Complex Test showcases one of the most important features of Dapper, which is the ability to modify the dataflow graph at runtime. With subflow embedding, as we call it, the embedding node "Fan" shown in Figure 2.5 is replaced with a subflow consisting of three "Debug" nodes in Figure 2.6 that form a simple fan-out followed by fan-in. Although the above process may seem ill-defined, we have taken great care to design subflow embedding as a feature that is well-specified, robust, and useful, and invite the reader to peruse

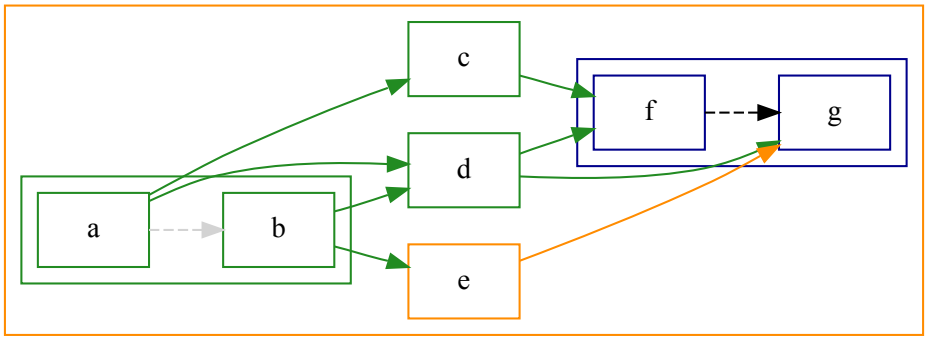


Figure 2.4: *The Simple Test.*

Section 3.2.2. To complicate matters, the test ends with a StreamEdge-induced connected component containing an "Error" node that, with some probability, simulates execution failure (an uncaught exception that propagates beyond the Codelet#run scope). Should an artificial error ever occur, the Dapper server will gracefully stop the "Error" computation and re-execute it until success. To try out the Complex Test, simply select the "ex.ComplexTest" Execution Archive Tree element and press the "run" button.

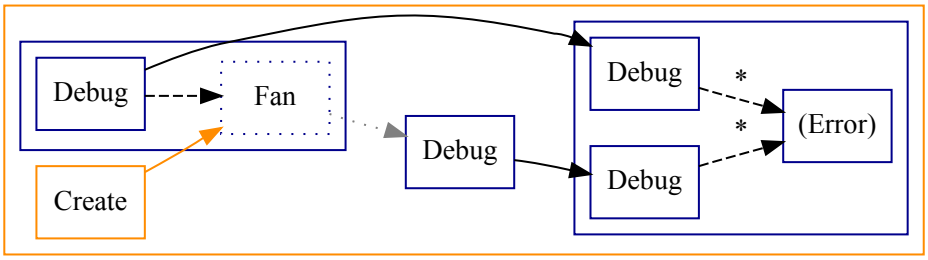


Figure 2.5: *The Complex Test before subflow embedding.*

2.3.3 Fork Bomb Test

In the Fork Bomb Test, we demonstrate the system's robust handling of large, complicated dataflow graphs that undergo modification at runtime. Very much like a traditional fork bomb, which attempts to overwhelm a host system with process fork requests, our (controlled) test uses subflow embedding to have a node propagate itself repeatedly up to a finite number of iterations. Thus, a seemingly trivial, benign dataflow consisting of a single "Fork Bomb" node can eventually grow into a monstrosity as in Figure 2.7. Inasmuch the Fork Bomb Test is fun to watch, it also tests the system's correctness with a combination of deliberately induced errors, large graphs, and complicated execution semantics. To try it out, simply select the "ex.ForkBombTest" Execution Archive Tree element and press the "run" button.

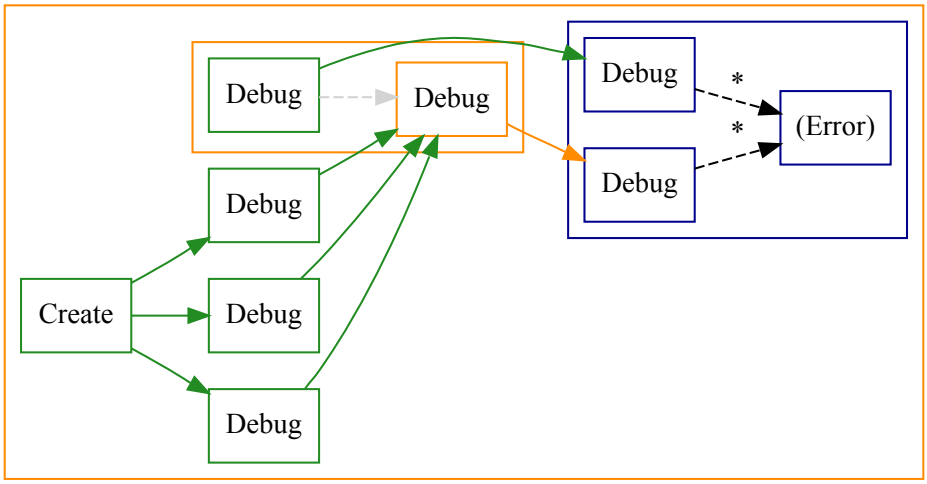


Figure 2.6: *The Complex Test after subflow embedding.*

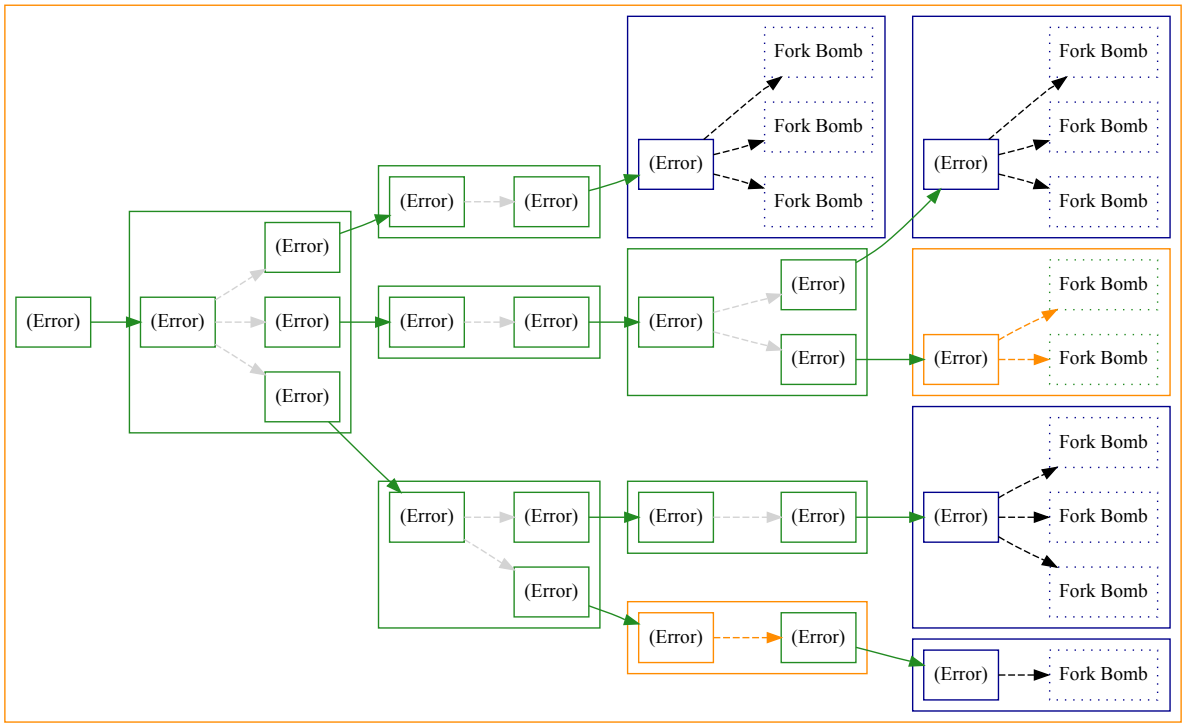


Figure 2.7: *The "Fork Bomb" node after embedding itself repeatedly.*

2.3.4 Merge Sort Test

As a proof of concept demonstrating how Dapper might see deployment, we introduce the Merge Sort Test. Although of little practical use, the task of sorting a large file in a distributed manner has a natural decomposition into smaller subtasks – those that slice up the file, those that sort the slices, and those that merge the results. In addition to selecting the "ex.MergeSortTest" Execution Archive Tree element and pressing the "run" button, you will also need start up at least 2^d remote clients, where d is the depth parameter described later. Please refer back to Section 1.3 if you have problems.

Upon initiating the Merge Sort Test, you will see a dialog from the user interface populated with fields. Fill them in as such:

1. depth – The depth parameter d mentioned above. Governs the amount of parallelization.
2. path – A networked file system directory you don't particularly care for that is readable and writable by all *remote* machines.
3. in – The input file name on the *local* machine.
4. input_size – The number of lines l. The input file will have size $1024 \times l$ bytes.
5. out – The output file name on the *local* machine.

Upon pressing the dialog's "ok" button, the Merge Sort Test will start running – first, the input file will be created at the "Create" node; next, the file will be uploaded to the remote machines on the cloud/grid and sliced according to the amount of parallelization; afterwards, the slices will be sorted; next, the results will go into a series of merges; finally, the result of the last merge will be downloaded from some machine on the cloud/grid. Notice the extensive use of streaming behavior here: All "Sort", "Merge", and "Download" nodes execute at the same time and communicate along $2^d - 1$ TCP streams.

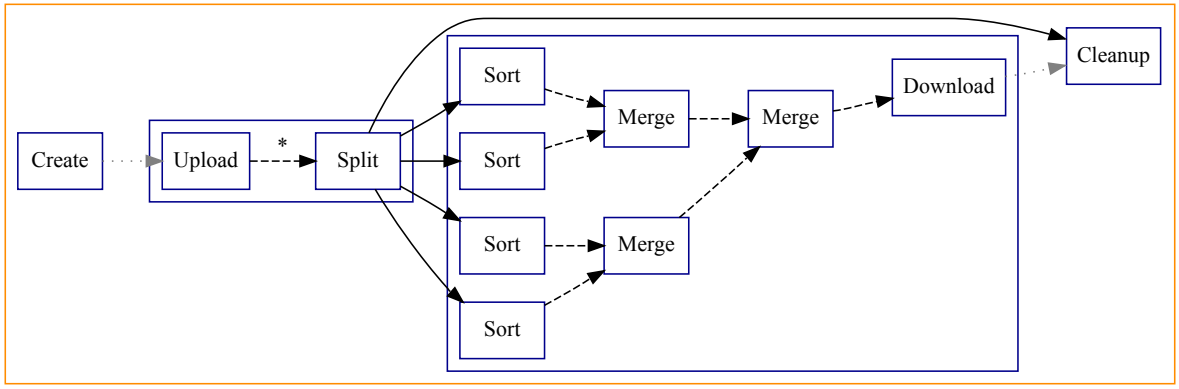


Figure 2.8: *The Merge Sort Test, a toy demonstration of Dapper's distributed operation over the cloud/grid.*

Chapter 3

The Dapper Programming API

After playing around with prefabricated examples, the user will invariably want to create original content. In anticipation of this, we offer a description of the Dapper programming API with three use cases in mind – writing programs as execution archives readable by the user interface; running the Dapper server as an embedded application (in Apache Tomcat, for example); and, for the bravest users, modification of Dapper itself. We start with a description of key class libraries, and go on to describe the advanced functionality that they make possible.

3.1 Core Classes

3.1.1 Basic Computation – Codelet and Resource

The Codelet interface encapsulates computations in a dataflow, and appear as nodes in the graph representation. When writing a codelet, one implements the Codelet#run method, which takes a list of input resources, a list of output resources, and a parameters DOM Node. In support of Codelet, the Resource interface is parameterized by the type of I/O object wrapped, be it an abstract data handle, InputStream, or OutputStream. The reader may think of codelets as callbacks – the types of resources passed in, as well as execution initiation, are controlled by the dataflow graph, whose construction we describe in Section 3.1.2. The code snippet below demonstrates a cleanup action implemented as a codelet, which iterates over the input resources and deletes each one in turn. The toString method is overridden so that "Cleanup" will appear if the name of the associated node is not explicitly set.

```
public class Cleanup implements Codelet {

    @Override
    public void run(List<Resource> inResources, List<Resource> outResources,
        Node parameters) {

        for (InputHandleResource ihr : CodeletUtilities.filter(inResources,
            InputHandleResource.class)) {

            for (String handle : ihr) {
                IoBase.delete(new File(handle));
            }
        }
    }
```

```
    /**
     * Default constructor.
     */
    public Cleanup() {
    }

    /**
     * Creates a human-readable description of this {@link Codelet}.
     */
    @Override
    public String toString() {
        return "Cleanup";
    }
}
```

3.1.2 Dataflow Construction – Flow and FlowBuilder

The Flow class holds bookkeeping information for dataflows. Instances of FlowBuilder, in the

FlowBuilder#build(Flow, List<FlowEdge>, List<FlowNode>)

method (assume the last two arguments are non-null empty lists for now), add FlowNodes and FlowEdges with the methods

Flow#add(FlowNode, FlowEdge...)

and

Flow#add(FlowEdge),

respectively.

The above methods are not without constraints, however, which maintain dataflow graph integrity. First, the node argument n to Flow#add cannot already exist. Second, all edge arguments must have the form (n', n), where $n' \neq n$ already exists. Third, the edge argument (u, v) to Flow#add must ensure that both u and v exist. Finally, the addition of an edge must **not** induce a cycle. For its own safety, the Dapper server will refuse to carry out actions that result in constraint violations. The code snippet below illustrates the usage of these methods.

```
public class SimpleTest implements FlowBuilder {

    ...

    @Override
    public void build(Flow flow, List<FlowEdge> inEdges, List<FlowNode>
        outNodes) {

        flow.setAttachment(flow.toString());

        FlowNode dn = new FlowNode("ex.Debug") //
                .setDomainPattern(LOCAL);

        FlowNode a = dn.clone().setName("a").setAttachment("a");
        FlowNode b = dn.clone().setName("b").setAttachment("b");
```

```
        FlowNode c = dn.clone().setName("c").setAttachment("c");
        FlowNode d = dn.clone().setName("d").setAttachment("d");
        FlowNode e = dn.clone().setName("e").setAttachment("e");
        FlowNode f = dn.clone().setName("f").setAttachment("f");
        FlowNode g = dn.clone().setName("g").setAttachment("g");

        flow.add(a);
        flow.add(b);
        flow.add(new StreamEdge(a, b));
        flow.add(c, new HandleEdge(a, c));
        flow.add(d, new HandleEdge(a, d), new HandleEdge(b, d));
        flow.add(e, new HandleEdge(b, e));
        flow.add(f, new HandleEdge(c, f), new HandleEdge(d, f));
        flow.add(g, new HandleEdge(d, g), new HandleEdge(e, g));
        flow.add(new StreamEdge(f, g));
    }

    ...
}
```

3.1.3 Declaring Computation – FlowNode

While Section 3.1.2 describes how to build dataflow graphs, it does not elucidate on where nodes come from in the first place. To create a new FlowNode, one simply instantiates the FlowNode(String) constructor with the desired (fully qualified) codelet class name. This will only work in the context of the FlowBuilder#build method's invoking thread, as it depends on a special class loading environment. One can then set multiple execution constraints with a builder-like pattern on the newly created FlowNode via the following methods:

- setRetries(int) – Sets the number of failed execution retries on this node before the server purges the entire dataflow.
- setTimeout(long) – Sets the timeout in milliseconds. When a computation times out, the server purges the entire dataflow.
- setParameters(Node) – Sets the parameters DOM tree, which clients executing this codelet will observe. Parameters are vital for dataflow initialization; without them, nodes with zero in-degree would not be able to locate their inputs.
- setDomainPattern(String) – Compiles and sets the regular expression Pattern for domain matching, effectively controlling what gets executed where. See Section 3.2.1 for a detailed description.
- setName(String) – Sets the name of this node for later reference via getName().

The code snippet below illustrates FlowNode creation in the FlowBuilder#build routine of a distributed merge sort.

```
public class MergeSortTest implements FlowBuilder {

    ...

    @Override
    public void build(Flow flow, List<FlowEdge> inEdges, List<FlowNode>
        outNodes) {
```

```
        FlowNode createSortFileNode = new FlowNode("ex.CreateSortFile") //
                .setDomainPattern(LOCAL) //
                .setParameters(String.format("<file>%s</file><lines>%d</lines>
                   ", //
                        this.inFile.getAbsolutePath(), //
                        this.inputSize));

        FlowNode uploadNode = new FlowNode("ex.Upload") //
                .setDomainPattern(LOCAL) //
                .setParameters(this.inFile.getAbsolutePath());

        FlowNode splitNode = new FlowNode("ex.Split") //
                .setDomainPattern(REMOTE) //
                .setParameters(this.pathDir.getAbsolutePath());

        FlowNode sortNode = new FlowNode("ex.Sort") //
                .setDomainPattern(REMOTE);

        FlowNode mergeNode = new FlowNode("ex.Merge") //
                .setDomainPattern(REMOTE);

        FlowNode downloadNode = new FlowNode("ex.Download") //
                .setDomainPattern(LOCAL) //
                .setParameters(this.outFile.getAbsolutePath());

        FlowNode cleanupNode = new FlowNode("ex.Cleanup") //
                .setDomainPattern(REMOTE);

        ...
    }

    ...
}
```

3.1.4 External Interface – Server, FlowProxy, Client

The Dapper Server class may be instantiated quite simply with its Server(InetAddress, int) constructor, whose arguments are a local address and the listening port number. In addition to binding a listening port for incoming client connections, the server also spawns multiple service threads. The server may be closed and its underlying resources freed at anytime with a call to Server#close.

The server's most important aspect is its ability to create dataflows with the Server#createFlow(FlowBuilder, ClassLoader) method, whose return value is an instance of the FlowProxy class. The first argument is the FlowBuilder that will construct the desired dataflow, while the second argument is the ClassLoader that has access to codelets requested by FlowNode instantiations. Since correctly providing the latter argument requires knowledge of Java class loading, a notoriously tricky topic, we defer further discussion to Section 3.4, and remark that operating the user interface does *not* require any knowledge of the above internals.

As mentioned earlier, for safety reasons, users do not have access to the server's internal Flow data structure directly, and instead interact with its proxy, which exports the methods await (wait until finished, InterruptedException for an interrupt, ExecutionException for an error); refresh (refresh the state); purge (stop the flow and release

associated clients); and finally toString, which emits a Dot-readable visual representation. The code snippet below illustrates a build-await-refresh loop.

```
// FlowBuilder fb;
// Server server;

FlowProxy fp = server.createFlow(fb);

fp.refresh();

displayDot(fp.toString());

fp.await();
```

The Dapper Client class complements the server with the computational wherewithal to execute dataflows. Its constructor,

Client(InetSocketAddress, String),

takes two arguments – the first denotes the server hostname to initially contact, and the second designates some user-defined domain, as described in Section 3.2.1. Clients are self-contained and do not require further interaction after being started from the command line by the ClientDriver class. With no user-provided arguments, the server address (–host) is assumed to be "<localhost>:<default dapper port>" and the domain (–domain) is assumed to be "remote". The port suffix of the server address is optional, and it defaults to Constants#DEFAULT_SERVER_PORT.

Under Dapper's client lifecycle policy, client processes live as long as their TCP connections to the server. It remains the user's responsibility to maintain a work pool. To assist in the management of clients, the server supports the Server#closeIdleClients() and Server#setAutocloseIdle(boolean) methods, which shut down idle clients – the former is a one-time call, while the latter modifies the server's behavior. To clarify, we define the idle number as that which is in excess of the total number of jobs immediately dependent on those currently executing. In other words, the server is looking ahead to the next batch of computations and gauging the amount of excess compute power.

3.1.5 Edge Types – HandleEdge and StreamEdge

Just as the nodes represent computation, edges of a dataflow graph represent data transfer. Dapper currently supports two edge types to facilitate passing of information from one stage of computations to the next in the form of user-interpreted abstract data handles and TCP streams. These are manifested as instances of the HandleEdge and StreamEdge classes, respectively, which both descend from the FlowEdge interface. A combination of edge types helps determine execution semantics – nodes connected by a HandleEdge execute sequentially, as we expect the upstream node to have produced a permanent input referenced by some data handle for the downstream node to read; on the other hand, all nodes in the connected component induced by StreamEdges execute simultaneously. As a consequence, execution failure semantics differ, too – one may consider abstract data handles as checkpoints of work completed, where, should downstream nodes fail, upstream nodes need not be re-executed; however, this is no longer true for stream transfers.

While the user may specify any acyclic dataflow graph consisting of exclusively HandleEdges, sometimes system specifications require that some data never touch disk and, as a consequence, that StreamEdges be used. To enable unambiguous execution semantics, we impose one additional constraint on the user – that for all connected components $C \subseteq V$ induced by stream transfers, there does **not** exist a HandleEdge (u, v) such that $u, v \in C$. In other words, we disallow connected components that are not internally consistent. To illustrate, consider Figure 3.1-a and 3.1-b. In 3.1-a, the interpretation is clear: Nodes $\{1, 2\}$ execute simultaneously first, followed by nodes $\{3, 4\}$ (blue rectangles delimit induced connected components). On the other hand, 3.1-a is rather ambiguous: All nodes must execute simultaneously, and yet, contradictorily, Node 4 depends on a file input from Node 2. Thus, when building a dataflow

graph, one need only keep in mind two simple rules: first, that the graph contains no (directed) cycles; and second, that all connected components contain no internal, non-stream edges. We also provide the DummyEdge type, which has no special behaviors or semantics other than to introduce dependencies. To declare edges, one simply invokes

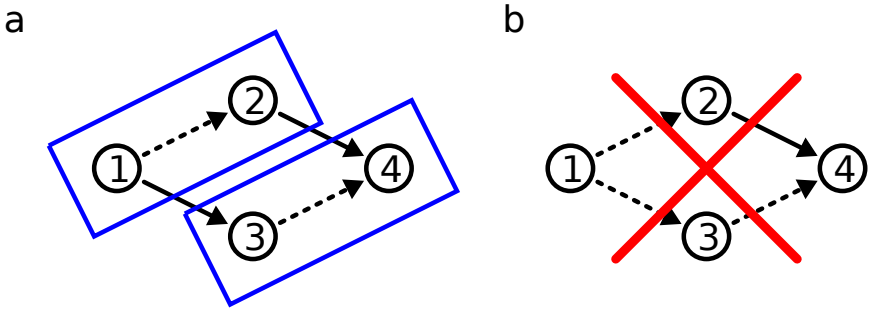


Figure 3.1: *An example of valid and invalid configurations for dataflows involving StreamEdges. Blue rectangles denote inferred connected components.*

the constructor of the desired edge type with the start and end FlowNodes. One can then use setters to customize class-specific properties listed in Table 3.1.

operation	class	description
setName(String)	FlowEdge	Sets the name retrievable by getName().
setInverted(boolean)	StreamEdge	Inverts the "connects to" relation among upstream and downstream nodes. See Section 3.2.1.
setExpandOnEmbed(boolean)	HandleEdge	If the end node embeds a subflow, this edge of n handles will be replaced with n edges of one handle each. See note in Section 3.2.2.

Table 3.1: *The properties supported by various types of edges.*

During codelet execution, HandleEdges and StreamEdges appear as their resource proxies in the form of InputHandleResource and StreamResource<InputStream> for inputs, and OutputHandleResource and StreamResource<OutputStream> for outputs. Note that StreamResource is parameterized by the type of stream carried, whether it be an InputStream or OutputStream. Manipulating StreamResources are as easy as getting their input/output streams and reading/writing on them. InputHandleResource and OutputHandleResource, however, operate like message queues – they export a series of getter operations for retrieving data handles and "stems" from upstream computations and putter operations for sending data handles and stems to downstream computations. While the meaning of handles and stems are open to interpretation by the user, consider a simple example of passing along files in the code snippet below.

```
// InputHandleResource ihr;
// OutputHandleResource ohr;

String filename = ihr.getHandle(0);
String path = derivePath(filename);
```

```
String stem = ihr.getStem(0);

// Do something.

ohr.put(String.format("%s/%s.out", path, stem));
```

The code above says:

> Interpret the handle at entry 0 as a filename (say, foo/bar/baz.in), and derive its path and extension; also remember the stem at entry 0 (say, baz). Afterwards, run a process that takes files of the form foo/bar/baz.in and creates outputs of the form foo/bar/baz.out. Finally, add a new handle entry foo/bar/baz.out with stem baz.

In general, handles reference the data created, and stems identify them in an way that is separate from superficial differences like path and extension in the file case. Table 3.2 provides a listing of supported methods. In addition to

operation	class	description
getInputStream()	StreamResource	Gets the underlying input stream.
getOutputStream()	StreamResource	Gets the underlying output stream.
getHandle(int)	InputHandleResource	Gets the handle at the given entry index.
getStem(int)	InputHandleResource	Gets the stem at the given entry index.
get(int)	InputHandleResource	Gets the entry at the given index as a String[] of the form [handle, stem].
get()	InputHandleResource	Gets the table of entries as an ObjectArray<String> of dimensions $2 \times n$, where handles and stems form the first and second columns.
iterator()	InputHandleResource	Creates an Iterator<String> over handles.
put(String)	OutputHandleResource	Adds a handle with the empty string stem.
put(String, String)	OutputHandleResource	Adds a handle and stem.
put(ObjectArray<String>)	OutputHandleResource	Adds all entries contained in an ObjectArray<String> of dimensions $2 \times n$.

Table 3.2: *The methods supported by various resources.*

member methods on resources, we also provide a collection of static methods for convenience in the CodeletUtilities class. They are listed in Table 3.3.

operation	description
groupByName(List<Nameable>)	Groups a list of resources by name.
filter(List<Nameable>, String)	Filters a list of resources based on name.
filter(List<Nameable>, Class)	Filters a list of resources based on class.
filter(List<Nameable>, String, Class)	Filters a list of resources based on name and class.
createStem()	Retrieves a stem from the server that is guaranteed to be unique over the server's lifetime.
createElement(String)	Interprets the given string as XML sandwiched between <parameters></parameters> tags.

Table 3.3: *The convenience methods provided by CodeletUtilities.*

3.2 Advanced Features

3.2.1 StreamEdge Inversion and Execution Domains

One of the most interesting interplays among Dapper's features is use of execution domains on FlowNodes combined with inverted connections on StreamEdges to traverse firewalls; hence, it would make sense to discuss them in the same context.

In many academic computing environments, the cloud/grid sits behind some kind of firewall. Assuming that the Dapper server does not reside behind a firewall itself, and that the existing firewall allows the establishment of outgoing TCP connections, but not vice versa, interoperability is still possible. With the

StreamEdge#setInverted(boolean)

method, one can control the directionality of the "connects to" relation among two nodes connected by a StreamEdge – in other words, if the machine bound to the upstream node initiates a TCP connection with the machine bound to the downstream node (false), or the other way around (true).

In Figure 3.2, we show how to apply StreamEdge inversion in a commonly reoccurring configuration. Imagine that the black nodes represent cloud/grid computations and the red nodes represent computations local to the Dapper server. Stream edges to and from the cloud show up as dashed lines; consequently, a "*" denotes that the edge inverts the "connects to" relation. Furthermore, using the same notation as in Figure 3.1, we demarcate the connected components they induce by blue bounding boxes. Notice that, since all TCP connections between the Dapper server and the cloud are initiated by machines from within, the restrictions of the firewall are bypassed.

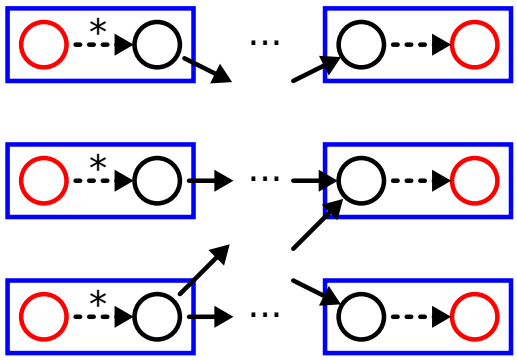


Figure 3.2: *A possible application of StreamEdge inversion.*

Up to this point, we did not discuss how execution domains play a role. With the

FlowNode#setDomainPattern(String)

method, the user can control what gets executed where; in the example above, it would be desirable that the black and red nodes were bound to remote and local machines, respectively. Recall from Section 3.1.4 that clients can take up to two arguments, the second of which specifies the domain. Thus, a client is considered eligible to execute at a node only if that node's domain pattern, given as a Java regular expression (see Pattern), matches. If not specified, a remote client's domain will default to "remote". Additionally, a client running on the same machine as the Dapper server will have its domain set to "local" no matter what. To relate back to the original example, setting the domain patterns "^remote$" and "^local$" for the black and red nodes, respectively, would be sufficient to distinguish among cloud machines and the local host.

3.2.2 Runtime Graph Modification – EmbeddingCodelet

For many dataflow programs, the final topology is not known at the start, and some computations may produce outputs that, upon inspection, determine the runtime topology. To this end, we introduce a special kind of codelet, EmbeddingCodelet, which, analogous to a function call, embeds a user-specified dataflow subgraph into the original graph. The EmbeddingCodelet interface extends both Codelet and FlowBuilder, since it computes embedding parameters on the client side and passes them to the embedding procedure on the server side. On top of the abilities bestowed by the above two interfaces, the class also exports the setEmbeddingParameters(Node) and getEmbeddingParameters() methods; client-side computations may message the server side with the former, and the server side may retrieve information derived on the client side (perhaps in a compute-intensive manner) with the latter.

Although the idea of runtime graph modification is not new, it has largely remained a performance optimization for the automatic parallelization of subtasks [4]. We treat subflow embedding, however, as a first-class construct that operates with a mechanism analogous to traditional function calls. Like function calls, we would want said feature to delineate an abstraction and safety boundary – that is, nodes not directly connected to an EmbeddingCodelet node should not need to know about its internals, and the server-side embedding code should not be able to corrupt the parent graph. By extending appropriate analogies to function calls in the ensuing sections, we seek to provide users with an intuitive and disciplined approach to an expressive component of the Dapper distributed programming model.

Consider the starred embedding node in Figure 3.3-a, and suppose that nodes 1, 2, and 3 have finished. Upon execution of the EmbeddingCodelet on the client side, the server invokes its

```
EmbeddingCodelet#build(Flow, List<FlowEdge>, List<FlowNode>)
```

method. One treats subflow embedding as if one were building an initial graph in Section 3.1.2, with the exception that the list of in-FlowEdges for the second parameter and the list of out-FlowNodes for the third require special treatment. Consequently, in-FlowEdges require assignment to newly added nodes via

```
FlowEdge#setV(FlowNode);
```

out-FlowNodes, on the other hand, do not require any sort of in-edge incidence. Note that out-FlowNodes, which are inferred via their out-neighbor relation with the embedding node, already belong to the parent Flow and *do not* require explicit addition via the Flow#add method. Figure 3.3-b depicts how the result might have looked. The dashed box delineates the embedding scope, which, in function call terminology, has the red edges incident on nodes 1, 2, and 3 as input parameters, and nodes 4, 5, and 6 as receivers for return values. Green edges and nodes are newly added. To further the analogy, since EmbeddingCodelets can embed themselves and others of their ilk, embedding scope is a stack frame of sorts.

The power and richness of subflow embedding comes with a few restrictions, caveats, and pitfalls. First, one cannot provide an invalid topology. On any violations, the server will immediately purge the parent dataflow and notify all attached clients. As an added self-protection measure, the server uses in-FlowEdge and out-FlowNode proxies to prevent tampering by users. Second, one must use DummyEdges to connect the embedding node with prospective out-FlowNodes, as in Figure 3.3-a. We impose this restriction to reduce confusion, since such edges, having served their purpose, are discarded afterwards. Third, to disambiguate among different types of edges and nodes while in EmbeddingCodelet#build, be sure to make use of FlowEdge#getName and FlowNode#getName, along with their corresponding setters. Fourth and finally, HandleEdges support special functionality with regards to embedding. Any HandleEdge with isExpandOnEmbed() true and a table of n entries (see Section 3.1.5), will expand into, or be replaced with, n HandleEdges each containing a singleton subtable.

For visual demonstrations of the mechanisms described above, please refer back to Sections 2.3.2 and 2.3.3.

3.3 Programming for the User Interface

Users that have gotten this far are no doubt eager to start running their own Dapper dataflows. Although entirely possible, deploying the Dapper server as an embedded application still requires a thorough understanding of the

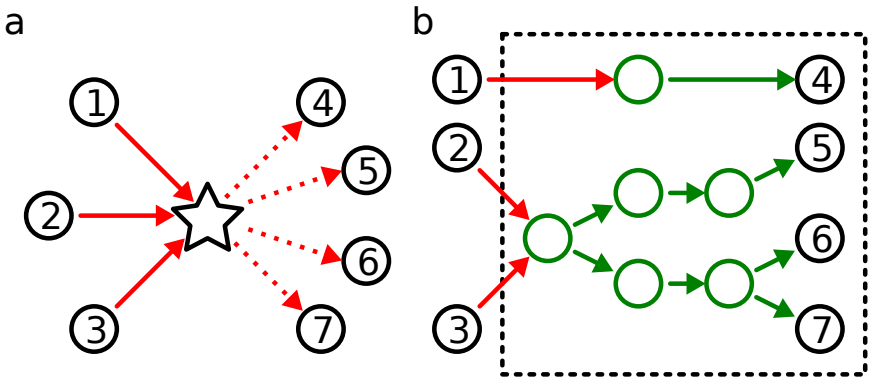


Figure 3.3: *A demonstration of subflow embedding before and after.*

Server class and related classes described in Section 3.1.4. For users wishing to see the quickest return on their time investment, however, we recommend following a few simple conventions and packaging an execution archive for dynamic loading by the user interface first described in Section 1.3.

As its name implies, an execution archive contains class files of executable Dapper code consisting of Codelet and FlowBuilder instances. While the user interface imposes no conventions on Codelet instances, it makes two requirements of FlowBuilders. First, the user makes a FlowBuilder visible to the user interface by marking it with the Program annotation. For this annotation, one can provide an optional array of strings (default is the empty array) that represent the argument names, which the user interface will prompt for as in Section 2.3.4. Second, every FlowBuilder **must** have a public constructor (see MergeSortTest(String[])) whose signature is exactly a singleton String[] standing for argument values.

Once you have packaged a Jar according to the above conventions, the user interface will scan it, extract (possibly multiple) Program-annotated FlowBuilder instances, and register them with the "Archive" tree. You will then be able to select a FlowBuilder of your liking and start its constructed dataflow with the "run" button shown in Figure 2.2-3. For your convenience, you may also invoke the user interface from the command line with the pattern

```
java -jar dapper.jar [--autoclose-idle] [--archive <archive> <builder> < \
   args...>],
```

where archive is the execution archive you wish to load, builder is the fully qualified class name of the desired FlowBuilder, and args... are the variable-length arguments to said FlowBuilder. Note that the optional –autoclose-idle argument toggles Server#setAutocloseIdle(boolean) underneath.

3.4 Programming for New Environments

In time, the user's needs may outgrow the user interface described in Section 3.3, and he or she may want to operate the Dapper server as an embedded application. The Server#createFlow(FlowBuilder, ClassLoader) method, which requires a ClassLoader as its second argument, presents the biggest technical hurdle – how does one even go about setting up an invocation? To answer this question, we briefly digress into a discussion of user interface internals, which contain the server as an embedded application.

By reading in execution archives, the user interface obviously makes their Codelet instances available for class loading by FlowNode instantiations in the FlowBuilder#build method. It makes extensive use of the Shared Scientific

Toolbox's RegistryClassLoader, which, as its name implies, relies on the ResourceRegistry abstraction. A registry is simply a place to look for class bytecodes. In particular, the user interface employs JarRegistry, which designates loaded Jar files as places to look for codelet classes.

Ideally, creating a ClassLoader with access to pertinent codelet classes should be no more complex than the internal mechanisms of the user interface. If you would like to learn about RegistryClassLoader as a possible solution, we invite you to study the src/org/dapper/ui/CodeletTree.java source code as well as documentation for the org.shared.metaclass package and its manual entry. Hence, it behooves us to think that embedding the Dapper server into a foreign environment is tantamount to understanding the basics of Java class loading.

Bibliography

[1] Jeffrey Dean and Sanjay Ghemawat. Mapreduce: Simplified data processing on large clusters. In *OSDI '04: Sixth Symposium on Operating System Design and Implementation*, 2004.

[2] Hadoop.

[3] Pig project.

[4] Michael Isard, Mihai Budiu, Yuan Yu, Andrew Birrell, and Dennis Fetterly. Dryad: Distributed data-parallel programs from sequential building blocks. In *European Conference on Computer Systems (EuroSys)*, pages 59–72, Lisbon, Portugal, March 21-23 2007.

[5] D Hull, K Wolstencroft, R Stevens, C Goble, MR Pocock, P Li, and T Oinn. Taverna: A tool for building and running workflows of services. *Nucleic Acids Research*, 34:729–732, 2006.

[6] Randal Bryant. Data-intensive supercomputing: The case for DISC. Technical report, Carnegie Mellon University, Pittsburgh, PA, USA, 2007.

[7] Ozalp Babaoglu, Lorenzo Alvisi, Alessandro Amoroso, Renzo Davoli, and Luigi A Giachini. Paralex: An environment for parallel programming in distributed systems. Technical report, Cornell University, Ithaca, NY, USA, 1991.