Building Dynamic, Long-Running Systems

Steven P. Reiss
Brown University
Context

Systems-of-Systems are common place

- Modern applications (waze)
- Applications using multiple data sources
- Applications using multiple back ends
- SoS will be the rule, not the exception
It should be easy to write and maintain such systems

- Handling failure
- Handling evolution
- Dealing with security, privacy, efficiency
- Handling data as well as control
- And not on an ad hoc basis
PROBLEM I

Systems are getting too big for one team to build everything.

- More reliance on open source solutions
- Reliance on outside services
- Crowd-sourced programming
- Make use of code already written
PROBLEM II

Long-running systems of systems make use of distributed components that both change and fail.

- Web services
- Micro services
- Remote calls
- Open source servers
- Phones and other devices
Applications should be able to make effective use of dynamically changing computing capabilities.

- Connections to servers
- Availability of local idle cycles
- Phones and other portable devices
- Automatic reconfiguration
Applications should be able to make use of the data available from today’s many devices.

- Phones
- Health monitors
- Emergency handling
- Cars
- Internet of things
OBJECTIVE

New ways of thinking about long-running programs built over distributed changing systems.

• Make them straightforward to code
• Handle failures, transient and permanent
• Handle evolution
• Handle data
Component-Based Programming

• **Appropriate Model for Complex Systems**
  • Component can be a system
  • Component can be a piece of a system
  • Component can be a library, class, …
  • Component can be data

• **Components**
  • Written by one programmer (or a team)
  • Accessible by others
  • Are independent of an application
Component-Based Programming

Provides a basis for systems of systems

- Natural hierarchical way of defining a system
- Failures = component failures
- Evolution = component evolution
- Security = component security
TAIGA (2003)

- Current Trends
  - Web services
  - Peer-to-peer computing
  - Grid computing
  - Common platforms
  - Open source
  - Browser-based applications

- What is the logical progression of combining these?
One World, One Program

• Everything is connected
• Programs communicate to get work done
• Processing is distributed
• Programs depend on each other’s data and computation

There is only one program
And it runs everywhere and all the time
Implications

• How do you write a “program”
• How to support large numbers of programmers who don’t trust each other
• Security and privacy
• What are the economics of programming
• Sharing data & files as well as code
• Device-independent user interfaces
• The environment is unstable
• How to scale Internet-size
TAIGA

A Framework for a “world program”

• Demonstrate feasibility
• Demonstrate scalability
• Provide solutions to the basic problems
  • How to program
  • How to accommodate multiple programmers
  • How to handle security & privacy
  • Handling failure and evolution
  • Shared data, UI, code, computation, …
  • Making it work economically
Outerfaces

- Interface to a component
  - Java interface syntax
  - Functions, internal data types, static methods
  - Constructors as default factory
- Semantics of the component
  - Test cases
  - Contracts
- Other constraints
  - Cost model
  - Security model
  - Recovery model
Outerface Example

touterface edu.brown.cs.newsview.taiga.NewsParser {
  import java.util.Map;
  description {{
    This outerface parses a URL to determine the country or countries (or
    state or states) that are the topics of the corresponding stories
  }}
  trait { rebind = true; }
  class Parser {
    public static ValueMap scanUrl(String url);
  }
  interface class ValueMap {
    public Map<String,Number> world_values;
    public Map<String,Number> state_values;
  }
  testcase test0 {
    ValueMap rslt = Parser.scanUrl("http://www.nytimes.com/...");
    assert(rslt.get("England") != null);
    assert(rslt.get("England") > 0.5);
  }
}
TAIGA

OUTERFACES

Package + Semantics
Package + Semantics

IMPLEMENTATIONS

Outerfaces + Code
Outerfaces + Code
Implementations

• Define a binding to an outerface
  • Can define multiple outerfaces
  • Does not have to be direct
    • Web service, RPC, External server, Library, …
  • Includes resource files

• Define constraints
  • How it can be used (binding models)
  • Who can use it
  • Security and privacy
  • Cost
implementation edu.brown.cs.newsview.taiga.QuickParser {

    import edu.brown.cs.newsview.qcrawl.QuickCrawlMap;
    import edu.brown.cs.newsview.qcrawl.QuickPageScan;

    resources "/u/spr/newsview" {
        "data/countries",
        "data/uscities",
        "data/usstates",
        "data/worldcities"
    }

    implements edu.brown.cs.newsview.taiga.NewsParser {
        using class Parser = edu.brown.cs.newsview.qcrawl.QuickPageScan;
        using interface class ValueMap = edu.brown.cs.newsview.qcrawl.QuickCrawlMap;
    }

    cost = 50;
}
TAIGA Network

- Peer-to-Peer backbone
  - Handles firewalls, failures, routing, …
  - Message-based, command-oriented
  - Simulated direct connections
  - Library system (offers and responses)
- Encrypted point-to-point communication
- Shared facilities
  - Distributed file access
  - Linda-like tuple space
CHALLENGES

• TAIGA provides a starting point
  • How to upgrade it to handle today’s systems of systems

• Handling Data as first class objects
  • Data can be generated by anyone
  • Data can be used as needed
  • Data sources will evolve
  • Data sources will come and go
Data Components

- Today’s systems depend on data
  - Waze, health data in an emergency, ...
  - Data is available in many forms
- Standardize data in terms of components
  - Data Interface describes the data
  - Data Provider implements that interface
DataFaces

• Syntactic Definition
  • Available fields (structure/table definition)
  • Filters, aggregations, ...

• Semantic Definitions
  • Units, consistency properties, ...

• Other Considerations
  • Costs
  • Security, privacy, ...

5/23/16
DataFace Example

dataface edu.brown.cs.loadview.taiga.MachineLoad {
    String host_name;
    String host_id;
    double load_average;
    long up_time;
    int num_process;
    long total_memory;
    long memory_used;
    long total_swap;
    long swap_used;

    units {
        up_time : minutes, total_memory : bytes, memory_used : bytes, total_swap : bytes, swap_used : bytes
    }
    restricts {
        0 <= load_average;
        0 <= up_time;
        0 <= num_process;
        0 <= total_memory;
        0 <= memory_used <= total_memory;
        0 <= total_swap;
        0 <= swap_used <= total_swap
    }
} // end of dataface MachineLoad
Data Provider

- Provides access to the data
  - Returns dataface-determined structure
  - Handles unit conversions, mappings, etc.
  - Filter determines applicability
- Multiple providers are supported
- Providers register with the system
Data Provider Example

dataface implementation edu.brown.cs.loadview.taiga.LinuxMachineLoad {

    application edu.brown.cs.loadview.impl.LinuxLoadChecker;
    using edu.brown.cs.loadview.impl.LinuxMachineLoad;

    implements edu.brown.cs.loadview.taiga.MachineLoad {
        using host_name = host_name;
        using host_id = host_id;
        using load_average = load_average;
        using up_time = getUpTime();
        using num_process = num_processes;
        using total_memory = total_memory;
        using memory_used = memory_used;
        using total_swap = total_swap;
    }

    units {
        up_time : seconds,
        total_memory : kilobytes,
        memory_used : kilobytes,
        total_swap : kilobytes,
        swap_used : kilobytes
    }

} // end of dataface implementation LinuxMachineLoad
Data Access

- Applications access datafaces by queries
  - Stream-based SQL language
  - Translated into FILTER/AGGREGATE
- Aggregation, filtering handled by system
  - Stream-based data processing
  - Client returned the aggregated fields
Sample Queries

- **Query**
  
  ```sql
  SELECT *
  FROM MachineLoad
  WHERE up_time > 30
  ```

- **Get the load structure from all machines**
  - Given machine has been up > 30 minutes
  - Gets the data as it is generated
Sample Query

- **COMPILED QUERY:**
  
  ```
  <DFACEQUERY WINDOW='60000' UID='sprtestquery1'>
    <ACTION TYPE='FILTER'>
      <FIELD MIN='5' METHOD='getUpTime' />
    </ACTION>
    <ACTION TYPE='AGGREGATE'>
      <GROUPBY METHOD='getHostName' SET='setHostName' VALUE='*' />
      <GROUPBY METHOD='getHostId' SET='setHostId' VALUE='*' />
      <COMPUTE METHOD='getLoadAverage' SET='setLoadAverage' OP='AVERAGE' />
      <COMPUTE METHOD='getUpTime' SET='setUpTime' OP='MAX' />
      <COMPUTE METHOD='getNumProcess' SET='setNumProcess' OP='SUM' />
      <COMPUTE METHOD='getTotalMemory' SET='setTotalMemory' OP='SUM' />
      <COMPUTE METHOD='getMemoryUsed' SET='setMemoryUsed' OP='SUM' />
      <COMPUTE METHOD='getTotalSwap' SET='setTotalSwap' OP='SUM' />
      <COMPUTE METHOD='getSwapUsed' SET='setSwapUsed' OP='AVERAGE' />
    </ACTION>
  </DFACEQUERY>'';
  ```

- **RESULT**
  - Single MachineLoad generated every 60 seconds
  - Ignore host, hostid; compute the rest
Data Processing

- Client
- Data Server
- Provider
- Data Server
- Provider
- Data Server
- Provider
Data Processing

- Make use of the underlying network
  - Sets up a tree of data servers
  - One or more per ring

- Servers create a tree for each query
  - Aggregation and filtering done locally
    - When possible
  - Timers + notification from children

- Treat the network as a stream processor
  - Stream query language (SQL-like)
On-Going Work

• Efficient Query Processing
  • Scalable

• Handling failure and evolution

• Security and Privacy
  • Data provider can limit access
    • Based on filter
    • Based on minimum aggregation count
  • Data provider can provide approximate results (Differential privacy)
Dynamic Long-Running Systems

Long-Running SoS Application

Outerfaces
Syntax + Semantics + Cost_Model + Security_Model

Automatic Dynamic Rebinding

Web Service
Open Source
Grid Service

Evolving, Dynamic Implementations

Datafaces
Syntax + Privacy_Model + Cost_Model + Security_Model

Data Query Processing

Evolving, Dynamic Data Sources
Taiga Futures

- Alternative semantic definitions
- Better cost models
  - Allow dynamic reconfiguration
- Better security models
- Enhanced binding models
  - RESTful interfaces, micro services
- Robustness and scalability
- Fully integrating data and control
- Where do we go from here?
Questions and Comments
Cost Model

- Takes multiple factors into account
  - Performance (on test cases) (CPU/memory)
  - Binding type (library, server, grid, web)
  - Traits
  - Cost of implementation
- Designed for extensibility
Security Model

- Based on Java Security Model
  - Defines what operations can/can’t be done
  - Files, sockets, system info, class loading, ...
- Validated when testing
  - Testing done in a sandbox environment
- Security context for library calls
  - Used to map resource files as well
- Security context for applications
  - Sandboxed when possible
Binding Model

• TAIGA binds implementations to outerfaces
  • Binding is an explicit operation
    • Requires passing the tests and constraints
    • Generates a saved version of the implementation
  • Done automatically on first use
• TAIGA finds implementations at run time
  • Using the economic model
  • Binds on the fly
• Same user code works for all bindings
  • Programmer codes to outerface
  • Downloaded library, server, web service, grid
Grid-Binding

- Finds a node to run the server on
  - Send out request to servers
  - With pertinent information
- Servers
  - Look at request and decide if they want it
  - Respond yes/no (or ignore)
- Binder chooses accepting server
  - Runs the service there
Type Model

- TAIGA maintains type consistency
  - Across implementations
  - Objects can be used with expected semantics
  - Collections are supported
    - Immutable if Java types
    - Mutable if TAIGA types
  - Types are mapped on calls and returns
- Makes coding remote applications easier
Failure Model

• Complex systems fail in different ways
  • Network failures
  • Server failures
  • System failures (wrong result, unexpected exceptions, contract failure, timeouts)
  • All can be viewed as component failures

• Application should continue working in the presence of failures
TAIGA Rebinding

- When an implementation fails
  - Either explicitly (call fails)
  - Or implicitly (contract fails, exception)

- TAIGA will rebind the outerface
  - Unbinds the original binding
  - Applies the cost model to find an implementation
  - Validates the new implementation
  - Binds the new implementation
Outerface Example

outerface edu.brown.cs.webview.taiga.WebManager {

description {{
    This outerface manages a set of files for the webview application, ensuring that they do not get too long. A transfer record is added to a file when it does exceed the 1M length limit
}}

trait { rebind=true; }

class FileManager {
    static public String getCurrentFile();
    static public String getFileForDate(long date);
}

testcase Test0 {{
    public static void test() {
        FileManager.getCurrentFile();
        TaigaTesting.success();
    }
}}

} // end of outerface WebManager
Outerface Example

outerface edu.brown.cs.newsviw.taiga.NewsCrawler {
  import java.util.Map;
  description {{ This outerface periodically crawls a particular web site for news. }}
  trait { rebind=true; }
  requires edu.brown.cs.newsviw.taiga.NewsParser;

class Crawler {
  public Crawler(String baseurl,int level);
  public void addRoot(String root);
  public void setValidEnds(String ends);
  public void addIgnoreLinkPattern(String pat);
  public void setHome(String home);
  public void setTimeLimit(long time);
  public void setBase(String base);
  public ResultMap getValues();
}

interface class ResultMap {
  public Map<String,Number> world_values;
  public Map<String,Number> state_values;
}

cost { bind : GRID >>= 1, SERVER >>= 4; }
Outerface Example

outerface edu.brown.cs.newsview.taiga.NewsClient {

description {{ ....... }}
import java.util.*;
trait { rebind=true; }
class Client {
    model { Map<String,Number> source_set }
    public Client() {
        model { source_set = new HashMap<String,Number>(); }
    }
    public void addSource(String name,double weight) {
        model { source_set.put(name,weight); }
    }
    public void removeSource(String name) {
        model { source_set.remove(name); }
    }
    public ClientValueMap getValues();
}
interface class ClientValueMap {
    public Map<String,Number> world_values;
    public Map<String,Number> state_values;
}
}
Implementation Example

implementation edu.brown.cs.webview.taiga.SimpleManager {
    using edu.brown.cs.webview.recorder.RecorderManager;
    implements edu.brown.cs.webview.taiga.WebManager {
        using class FileManager =
            edu.brown.cs.webview.recorder.RecorderManager;
    }
    cost = 40;
    available *;
}
Security Extensions

• Peer-to-peer backbone has to be secure
  • Clients are who they say they are
  • Clients are running proper code
  • Clients are limited to particular domains
  • Add a notion of identity

• Create a private version of TAIGA
  • In addition to the public, everywhere version
Semantic Definitions

- Test cases & contracts are limiting
  - Broader than formal specifications
  - But still difficult to define in many cases
- Going beyond test cases
  - Partial specifications
  - Pseudo-code, frameworks, sketches, ...
  - Interaction with the programmer
Deploying at Scale

- TAIGA is a prototype
  - P2P network needs work
  - Unbinding of libraries not clean
  - Sandboxed execution of tests
  - Can be much more efficient
  - No phone-based implementation

- Needs to work with 1000s of nodes
  - Only tested with ~100
  - Generally running with ~10