Creo: Reduced Complexity Service Development

Per-Olov Östberg and Niclas Lockner
Dept. of Computing Science, Umeå University, Umeå, Sweden

Keywords: Service-orientated Architecture, Service Development Tools.

Abstract: In this work we address service-oriented software development in distributed computing environments, and investigate an approach to software development and integration based on code generation. The approach is illustrated in a toolkit for multi-language software generation built on three building blocks; a service description language, a serialization and transport protocol, and a set of code generation techniques. The approach is intended for use in the eScience domain and aims to reduce the complexity of development and integration of distributed software systems through a low-knowledge-requirements model for construction of network-accessible services. The toolkit is presented along with a discussion of use cases and a performance evaluation quantifying the performance of the toolkit against selected alternative techniques for code generation and service communication. In tests of communication overhead and response time, toolkit performance is found to be comparable to or improve upon the evaluated techniques.

1 INTRODUCTION

Cloud computing has in recent years evolved to an established paradigm for provisioning of IT capacity. While this approach can offer several benefits compared to traditional static provisioning, e.g., facilitation of more flexible service types (Armbrust et al., 2010) and improvements in cost and energy efficiency of large-scale computing (Walker, 2009; Berl et al., 2010), it also places focus on a current problem in distributed computing: the increasing complexity of development and management of systems in distributed computing environments (Kephart and Chess, 2003).

Service-Oriented Computing (SOC) is a popular approach to software development and integration in large-scale distributed systems. SOC is argued to be well suited for cloud environments as it places focus on representation of logic components as network-accessible services, and aims to facilitate development and integration of systems through coordination of service interactions. At architecture level, Service-Oriented Architectures (SOAs) define service interfaces as integration points and address system composition at interface or protocol level. While a number of SOA techniques have emerged, service development and integration are still complex issues and there exists a need for development tools that provide non-complex and low-learning-requirement environments for efficient development of service-based systems.

To illustrate these issues, we here take the perspective of eScience application development. In eScience[1], distributed computing techniques are used to create collaborative environments for large-scale scientific computing. In comparison to commercial software stacks, scientific computing tools are typically prototype-oriented, developed in projects with limited software development budgets, and often composed of heterogeneous components developed in multiple languages and environments. In addition, eScience applications often use distributed or parallel programming techniques to exploit the inherent parallelism of computational problems. As many current eScience efforts are approaching construction of virtual infrastructures using cloud technology, they here serve as illustrative examples of the difficulties of developing multi-language software stacks in heterogeneous distributed computing environments.

In this work we address reduction of complexity in service-based software development, and present an easy-to-use toolkit for efficient cross-language integration of software services. The toolkit is based on three core components: a simplified syntax service description language, a transparent data serialization and transmission protocol, and a set of code generation tools designed to abstract complexity in service and service client development.

[1]Computationally intensive science carried out in highly distributed network environments.

The remainder of the paper is structured as follows: Section 2 presents project background and a brief survey of related work, Section 3 outlines the proposed approach and toolkit, and Section 4 discusses use cases for the approach. In the second half of the paper, Section 5 contains a performance evaluation quantifying toolkit performance against selected alternative techniques for code generation and service communication, followed by conclusions and acknowledgements in sections 6 and 7.

2 RELATED WORK

A number of tools for service development and middleware construction exist, ranging in complexity and abstraction levels from very simple fine-grained interprocess communication tools to advanced middleware construction tools featuring advanced data marshalling, call translation, and remote reference counting techniques. In general there exists trade-offs between complexity and efficiency that make service technologies more or less suitable for certain situations, and many technologies have been developed for specific application scenarios.

For example, direct interprocess communication technologies such as traditional remote procedure calls (RPC) (Birrell and Nelson, 1984) and Java Object Serialization (JOS) (Oracle, 2005) (over sockets) provide transparent development models but offer little in ways of complexity abstraction. Other approaches such as Java Remote Method Invocation (RMI) (Wollrath et al., 1996) and the Microsoft Windows Communication Framework (WCF) (Mackey, 2010) offer development models tightly integrated into mature commercial software development environments, but lose some applicability in multi-platform application scenarios. There exists also standardized approaches to multi-language and multi-platform service development, e.g., the Common Object Request Broker Architecture (CORBA) (Vinoski, 1993), but while such standardized approaches typically are very expressive and capable of application in multiple programming styles, e.g., object-orientation and component-oriented development, this general applicability often comes at the price of very steep learning curves and high development complexity.

In service-oriented computing and architectures, programming models such as SOAP and REST-style web services are widely used due to features such as platform independence, high abstraction levels, and interoperability. The SOAP approach to web services favors use of standardization of XML-based service description and message formats to facilitate automated generation of service interconnection code stubs, dynamic service discovery and invocation techniques, and service coordination and orchestration models. SOAP-style web services are however often criticized for having overly complex development models, inefficiencies in service communication, and low load tolerances in servers (although developments in pull-based parser models have alleviated some of the performance issues (Govindaraju et al., 2004)).

The REpresentational State Transfer (REST) (Fielding, 2000) web service model is often seen as a light-weight alternative to the complexity of SOAP-style web service development. The REST approach discourages standardization (of message formats), promotes (re)use of existing wide-spread technology, and aims to give service developers more freedom in, e.g., choice of data representation formats and API structures. While this approach facilitates a development model well suited for smaller projects, it is sometimes argued to lead to more tightly coupled service models (that require service client developers to have knowledge of service-side data structures) and introduce technology heterogeneity in large systems.

Although service models are considered suitable for large-scale system integration, and some understanding of the applicability of web services has been gained (Pautasso et al., 2008), neither approach fully addresses the requirements of service-oriented software development and a number of technologies for hybrid service-RPC mechanisms have emerged. These include, e.g., interface definition language (IDL) based technologies such as Apache Thrift (Slee et al., 2007), an RPC framework for scalable cross-language service development, Apache Avro (Apache, 2009), a data serialization system featuring dynamic typing, and Google protocol buffers (Google, 2008), a method for serializing structured data for interprocess communication. For high performance serialization and transmission, there also exists a number of non-IDL based serialization formats and tools such as Jackson JSON (Jackson, 2009), BSON (MongoDB Inc., 2007), Kryo (Kryo, 2009), and MessagePack (Furuhashi, 2011).

In addition to trade-offs for technical performance and applicability, tools and development models often impose high learning requirements in dimensions orthogonal to the task of building distributed systems. For example, the Distributed Component Object Model (DCOM) requires developers to understand data marshalling and memory models, Java RMI distributed garbage collection, CORBA portable object adapters (type wrappers), and SOAP web services XML Schema (for type definition and validation). As distributed systems are by themselves complex to develop, debug, and efficiently analyze, there exists a need for software development tools that provide transparent and intuitive development models, and impose low learning requirements.

In this work we build on the service development model of the Service Development Abstraction Toolkit (Östberg and Elmroth, 2011), and investigate an approach to construction of development tools focused on reducing complexity of service-based software development. The aim of this approach is to combine the high abstraction levels of SOAP-style web services (using a simplified service description syntax) with the communication efficiency of more direct RPC-style communication techniques, and produce tools with low learning requirements that efficiently facilitate service development. As the work is based on code generation, the approach can be seen akin to development of a domain-specific language (Van Deursen et al., 2000) for service description, but the main focus of the work is to reduce overhead for exposing component logic as network-accessible services. The work is done in eScience settings, and presented results are primarily intended to be applied in scientific environments, e.g., in production of tools, applications, and middlewares for scientific simulation, experimentation, and analysis.

3 CREO

Service-oriented architectures typically expose components and systems as platform independent, network-accessible services. While this approach gracefully abstracts low-level integration issues and provides for high-level architecture design models, it can often lead to practical integration issues stemming from, e.g., complexity in service development models, steep learning curves of service development tools, and lack of distributed systems development experience in service client developers.

In this paper we build on earlier efforts presented in (Östberg and Elmroth, 2011) and (Östberg et al., 2012), and propose an approach to service development that places the responsibility of service client development on service developers. As this shift in responsibility introduces noticeable additional complexity in service development, e.g., in requirements for multi-language service client development, we note a need for tools to support the approach and present *Creo* - a service development toolkit based on automated code generation.

The Creo toolkit is aimed to reduce complexity in construction of network-accessible services by providing a development model that lowers learning requirements and increases automation in service development. While the toolkit is designed to be simple to use and targeted towards developers with limited distributed systems development experience, it also strives to provide service communication performance high enough to motivate use of the toolkit in mature service development scenarios.

To limit the scope of the work, we have initially designed the toolkit to support development of services in a single language (Java), and service client development in four languages common in eScience environments: C, C#, Java, and Python. The toolkit implementation patterns are however transparent and modularized, and all modules are designed to be extensible to code generation in additional languages. The intent of the toolkit is to provide robust service communication stubs in general purpose programming languages that can later be used to build integration bridges into special purpose environments such as R and Matlab. The choice of Java as service language is motivated by the language's rich development APIs, robustness in performance, platform independence, and wide-spread adoptance in operating systems and server platforms. The design philosophy of the toolkit can be summarized as supporting advanced implementation of services while keeping generated code for clients as transparent, lightweight, and free of external dependencies as possible.

To combine the ease-of-use of high abstraction level tools with the communication performance of more fine-grained approaches, the toolkit development model is based on the service description approach of SOAP-style web services combined with a customized version of the RASP protocol presented in (Östberg et al., 2012). The toolkit service development process can be summarized in three steps:

1. Service description. Service type sets and interfaces are defined in a custom service description (interface definition) language

2. Communication code generation. Service and service client communication stubs are generated from service descriptions.

3. Service integration. Logic components are exposed as services through implementation of generated service interfaces, and service clients are implemented based on the generated communication stubs for service interconnection.

In all steps of this process, the toolkit aims to reduce the complexity of service development by providing intuitive tools and formats for service description, data representation, and code generation.

3.1 Service Description

Program 1 A sample Creo service description.

```
// annotations
@PACKAGE("packagename")

// type definitions
struct MetaData
{
  String description;
  long timestamp;
}

struct Data
{
  MetaData metadata;
  double[] samples;
}

// interface definitions
interface DataService
{
  void storeData (Data[] data);
  Data retrieveData (String description);
}
```

For data type and service interface definition, the toolkit employs a service description language comprised of three parts:

- Annotations. Define code generation parameters, e.g., service package names.
- Types. Specifies a basic set of primitive types and a struct mechanism for type aggregation.
- Interfaces. Define service interfaces in terms of methods and method parameters.

The service description language format is based on the block syntax of the C/C++ family of languages. In the interest of simplicity, the primitive type set is restricted to a basic type set commonly occurring in most programming languages: `byte`, `char`, `int`, `long`, `float`, `double`, and `String`. The language supports direct aggregation of primitive types in structs and arrays as well as construction of compound types via aggregation of structs. This allows construction of hierarchical data types such as trees, but not cyclic data types such as graphs. Program 1 contains a sample service description demonstrating the aggregation mechanisms of the Creo service description language.

While alternative representation formats with more advanced features exist, e.g., schema-based type set and data validation in XML and WSDL, the design philosophy of this work is to reduce complexity rather that offer advanced features. The goal of the description language is to provide a convenient format that has great expressive power, is as unambiguous as possible, and introduces as few learning requirements as possible. The primitive type set defined, as well as the concept of aggregation of fields in records and arrays, are prevalent in programming languages and should prove intuitive to developers regardless of background. To minimize the learning requirements of the tool, the type interpretations and syntax of the description language are based on a subset of the well-known Java programming language.

3.2 Data Representation

To promote transparency, the representation format specified in service description also directly outlines the data structures used in data serialization and transmission. For language and platform independence, all values are transformed to and from network byte order in transmission and support code is generated for programming languages not supporting description language features (e.g., byte order transformation, string classes, or array types). For aggregated types, types are serialized in the order declared (and stored in memory), with size counters prefixing data for array types and strings. As data are declared and stored in hierarchical structures (trees), data serialization is essentially a left-wise depth-first traversal of data trees, where individual node values are stored sequentially. In terms of invocation semantics, Creo defines call-by-value semantics for invocation of remote service methods. As data are serialized by value, the use of reference and pointer types inside data blocks passed to services is not supported. In particular, use of circular references (e.g., cyclic graphs) may lead to inefficient transmission performance or non-terminating loops.

For efficiency in transmission (i.e. minimization of system calls and alignment of network package sizes to maximum transfer units), all data are serialized and deserialized via transmission buffers located in the generated code stubs. The protocol used for transmission of data between clients and services (illustrated in Figure 1) is a customized version of the Resource Access and Serialization Protocol (RASP) of the StratUm framework (Östberg et al., 2012). The description language does not support encoding of explicit exception messages for propagating error information across process boundaries.

3.3 Code Generation

Service integration code is typically provided in one of two forms: APIs or service communication stubs. To reduce complexity in service client development,

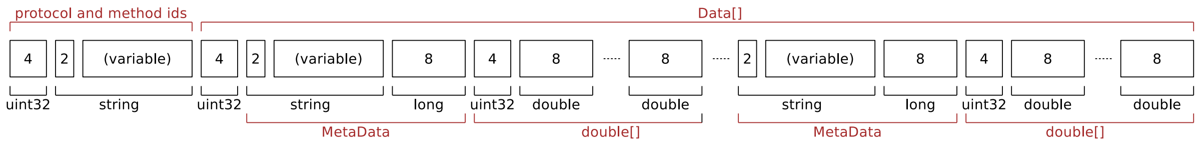


Figure 1: Byte layout of the Creo protocol request message for the `sendData()` method of Program 1. Data encoded in the order defined in service descriptions, arrays and strings prefixed with item counts. Byte block sizes and primitive types in black, protocol preamble (protocol and method ids) and aggregated (struct and array) types in red.

and increase the transparency of the service communication mechanisms, the Creo toolkit uses a code generation approach centered around immutable wrapper types and call-by-value interfaces. The rationale of this design is to make use of generated client code as intuitive as possible, and to facilitate a service client development model that doesn't require prior distributed systems development experience.

Use of code generation techniques rather than APIs fundamentally assumes that service descriptions rarely change (as service-oriented architectures tend to be designed in terms of service interfaces), and have the added benefits of allowing typed languages to catch type errors earlier while keeping service client implementations loosely coupled to services.

3.3.1 Code Generator

From a high level, the Creo toolkit can be seen to be composed of three components: a service description parser, a framework generator, and a custom package generator. To promote flexibility and facilitate adaptation to new requirements, e.g., support for new programming languages or representation formats, the architecture of the toolkit is designed to be modular and extensible. The separation of code generation for frameworks and custom packages (i.e. code specific to data types and services defined in service descriptions) serves to facilitate third party implementation of code generator plug-ins. With this separation it is possible to contribute plug-in modules to support alternative implementations of, e.g., data serialization routines and client implementations, without having to deal with generation of framework code.

The service description parser is constructed using a combination of in-memory compilation of the service description types (after replacing selected keywords to make service descriptions Java compliant), use of the Java reflection API (to validate description structures), and a custom language parser (that extracts parameter information). To isolate code generators from document parsing, the parser provides a full internal API that completely describes the type sets and document structures of service descriptions.

3.3.2 Generated Code - Framework

To establish a uniform model for client-service communication, all service client code implements a framework model for connection establishment, data serialization, and transmission capabilities. This framework is structured around an identified core feature set that includes, e.g., primitive type representation and serialization (including network byte order transformations), array and string type wrapper types (for languages not providing such types), and socket-level read and write transmission buffers.

The purpose of the framework is to separate service and client logic independent of the types and services defined in service descriptions, and reduce the complexity of generating code for service-dependent logic. Implementation of this framework pattern keeps all service client implementations lightweight and compatible with the service implementation, which facilitates development of client implementations in additional languages. On the service side, the framework code is connected to the service-dependent code through a provider-pattern implementation for service data type serializer factories.

3.3.3 Generated Code - Service Side

On the service side, the generated framework is extended with a lightweight service hosting environment containing basic server functionality such as thread and service management. The architecture of the service framework is based on the principle of abstracting as much as possible of the service boilerplate code required to expose components as services. It is the intent of the toolkit that service implementation should consist only of two steps - generation of the service framework from a service description file and implementation of a service (Java) interface.

The basic structure of the generated services is designed around the information flow in the system; a server hosts services, parses incoming requests, and passes request messages onto an incoming message queue for the requested service. The service implementation processes requests, generates and pushes response messages onto the outgoing message queue for the service. The server continuously monitors all service message queues and sends response messages when available. The core of the generated service framework is message-oriented and defined around the concept of asynchronous message queues, and does not restrict service implementations to use of only synchronous request-response communication patterns. However, while service implementations are free to define their own communication patterns in terms of the messages exchanged between clients and services, use of asynchronous communication patterns requires modifications of the generated service clients to fully support such exchanges. For reference, an asynchronous client (in Java) is provided with the generated service framework.

3.3.4 Generated Code - Client Side

The architecture of the generated service clients follows the same pattern in all implementing service client languages (C, C#, Java, and Python), and is designed to abstract fine-grained service communication tasks. A service API is generated exposing the methods defined in service descriptions, and all data are managed in immutable wrapper types based on the types defined in service descriptions. Service communication details, such as connection establishment and data marshalling, are abstracted by clients stubs.

The underlying philosophy of the toolkit is that it should be the responsibility of the service developer to provide integration code (service clients) and APIs for services, and the toolkit aims to abstract as much as possible of that process. To promote transparency, all client code generated is designed to follow the same design pattern and all generated service client code is designed to be as homogeneous as possible in architecture, code structure, and API functionality support. When applicable, all code is generated along with sample build environment data files (e.g., makefiles for C and ant build files for Java). In-memory compilation and generation of Java Archive (JAR) files are supported for Java.

4 USE CASES

To illustrate toolkit use, we here briefly discuss example application scenarios in the eScience domain. Envisioned use cases for the Creo toolkit include:

- Coordinated multi-language logging and configuration. Scientific applications in the eScience domain often consist of multiple components and systems developed in multiple programming languages. Coordinated logging of application state information can be very useful for visualization and management of application processes, which can be achieved by, e.g., developing a database accessor component in Java and exposing it as a service using the Creo toolkit. Client stubs generated by the toolkit can then be used to coordinate system logs from multiple sources without introducing external dependencies in systems. Similarly, multi-component systems can also use this technique to coordinate system configuration, allowing dynamic reconfiguration of systems (use cases from the StratUm (Östberg et al., 2012) project).
- Multi-component system integration. The Aequus system (Östberg et al., 2013)) system is designed for use in high performance and grid computing infrastructures. While the core of the system is developed in Java, the system also contains specialized components and tools developed in other languages, e.g., scheduler integration plug-ins in C and visualization and statistics tools in Python and Matlab. Use of the Creo toolkit allows smooth integration of different parts of the system without extensive distributed systems development effort.
- System evaluation experiments. Distributed computing infrastructure systems constructed as service-oriented architectures often require simulation experiments for testing and validation of functionality. The previously mentioned Aequus system is developed and evaluated using emulated system environments for system tests and scalability simulations. In these settings the Creo toolkit allows easy integration of multiple simulation components for surrounding systems (e.g., batch schedulers and accounting systems), and construction of large-scale emulation systems for system evaluation.
- Application cloud migration. Many eScience applications are initially developed for use on a single machine and later (for performance and scalability reasons) transformed into multi-component systems using parallel and distributed computing techniques. As part of this process, staging of applications into cloud environments often requires some form of reformulation of computational algorithms to better adapt to horizontal cloud elasticity models. The Creo toolkit can here be used to, e.g., build staging and monitoring tools or to facilitate remote communication with applications running in cloud data centers.

Use cases such as these illustrate not only the expressive power of tools for service development and component integration, but also the importance of keeping such tools simple and reducing the complexity of building distributed systems. Use of develop-

Table 1: A brief overview of the feature sets of the evaluated service technologies.

	Creo	Thrift	SOAP	REST	RMI
Interface type	IDL	IDL	IDL	protocol	API / stubs
Integration style	stubs	stubs	API / stubs	API / protocol	stubs
Data representation format	binary	text / binary	text	text / binary	binary

ment tools with steep learning curves or advanced knowledge requirements for, e.g., serialization formats, marshalling techniques, and transmission formats, can greatly add to the complexity of building distributed systems. For many purposes, and prototype development in particular, reduction of complexity and ease-of-use often outweigh the additional features of more advanced approaches.

5 EVALUATION

Service-based software design is an area with many competing approaches to service development and integration, making objective evaluation of new tools non-trivial.

In this work we identify three abstraction levels for development toolkits; low (fine-grained message level integration), intermediary (remote procedure call communication abstraction), and high (service-oriented component integration); and evaluate the proposed toolkit against selected tools from each abstraction level in the dimensions of serialization overhead, transmission overhead, and service response time. To facilitate future comparison against third party tools, we select well-established and easily accessible tools for the evaluation.

For low level abstractions we compare the performance of the toolkit against that of Apache Thrift (Apache, 2010), a software framework for scalable cross-language service development. As the toolkit primarily targets service development in Java, we have for high and intermediary levels selected Java-based tools. For intermediary level we evaluate two related technologies: Java Remote Method Invocation (RMI) (Wollrath et al., 1996), an object-oriented remote procedure call mechanism that supports transfer of serialized Java objects and distributed garbage collection, and Java Object Serialization (JOS) (Oracle, 2005), the object serialization technology used by Java RMI. For high level, we evaluate the toolkit against two popular web service technologies: REST web services (using the RESTlet framework version 2.0.15 (Restlet, 2013)) and SOAP web services (using the Apache Axis 2 SOAP framework version 1.6.2 (Apache, 2005)). Table 1 provides a brief comparison of the feature sets of the evaluated service technologies.

5.1 Testbed and Experimental Setup

To evaluate the technical performance of the toolkit we measure three facets of service communication performance; serialization overhead, transmission overhead, and response time; and quantify these against corresponding measurements of selected alternative tools. Serialization overhead is here defined in terms of the computational capacity used for generation and parsing of service messages, and is included in tests as it can heavily impact the execution footprint of service-based tools. Transmission overhead is here defined to be the additional bandwidth requirements introduced by service data representation formats, and is measured by quantitative comparison of total message sizes and message payload (raw data) sizes. To isolate the communication overhead components introduced by service tools in response time measurements, thin service implementations (minimal request processing times) are used.

Tests are performed using three types of request data; coarse-grained data (byte chunks), fine-grained number-resolved data (integer and float values), and fine-grained string-resolved data (text segments). For each test and request type, tests are performed with request sizes grown by orders of magnitude (blocks of 100, 1k, 10k, 100k, 1M, 10M and 100M bytes). Coarse-grained requests consist of large chunks of bytes without structured format. For clients based on Creo, Thrift, RMI, and JOS coarse-grained data are sent as raw byte arrays. For REST-based clients, requests are sent in HTTP POST requests as raw bytes with the MIME type ”application/octet-stream”. In SOAP-based clients, request data are encoded as Base64-encoded strings.

Data for fine-grained requests are created by grouping data in blocks of 10 bytes, grown by aggregating data blocks in groups of 10, and padded using smaller data blocks to align sizes with even exponentials of 2. For example, a 1k (1024 bytes) data block consists of 10 groups of 10 blocks of 10 bytes plus padding in the form of two 10 byte blocks and a 4-byte pad value (a 32-bit integer or a 4-byte string depending on type). Larger data blocks are grown using the same scheme, e.g., by aggregating ten 1k data blocks to form a 10k data block. Numbers-based data blocks contain pairs of 64-bit double-precision floating point and 16-bit integer values. String-based data blocks contain 10-character strings.

For serialization overhead and service response time tests, all tests are done by measuring the client-side makespans of full operations, starting at the point of client invocation and ending when the client receives a uniform size 4 byte server response message. To isolate overhead components, all measurements are performed in closed loop system settings using sequential invocation patterns on dedicated machines with no competing load and isolated network segments. Experiments are repeated multiple (at least ten) times to minimize the impact of external factors on measurements. Parallel invocation tests are used to evaluate the load sensitivity and scalability of service tools. All services used in measurements are implemented in Java and service clients are implemented in C, C#, Java and Python. For tests of the service response time of REST and SOAP tools, request serialization is done in JSON (using the reference library of json.org) and XML (using JAXB).

All tests are run on a dedicated symmetric cluster where nodes are equipped with dual 16 core 2.1 GHz AMD Opteron 6272 processors and 54 GB RAM. Nodes are interconnected with 1 Gbps links and networks are configured using MTU sizes of 1500 bytes. All nodes run Ubuntu Linux 12.04 kernel version 3.2, OpenJDK 1.6, Python 2.7, Mono 2.10, and GLib 2.32. All software are accessible from Ubuntu repositories.

5.2 Serialization Overhead

To isolate measurements of data serialization overhead, it is necessary to exclude all artefacts from transmission of data between clients and the servers in tests. Additionally, as tools employ transmission (read and write) buffers that consume computational power and are orthogonal to data serialization, transmission buffers need to be bypassed in tests. To quantify the serialization overhead of Creo and Thrift service clients, both generated code and runtime libraries are modified so that no data are placed in transmission buffers or sent to servers after serialization. Furthermore, both tool's service clients are modified so that they do not read data from servers after invocations.

To avoid modifications of Java RMI stacks, we here include measurements of the underlying serialization technology used (JOS) and assume measurements are representative of the serialization overhead of RMI. To quantify the serialization overhead of JOS, `ObjectOutputStream` instances are wrapped around non-buffered dummy output streams (no data transferred to underlying sockets). After modifications, serialization overhead tests are performed in the same way as service response time tests.

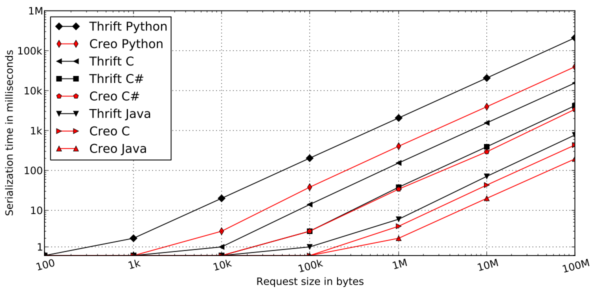
Results from data serialization overhead tests are visualized in figures 2 and 3. For ease of comparison, test results for multi-language tests (comparing Creo to Thrift) using fine-grained data tests are presented individually, separating tests using number-resolved and string-resolved data. As can be seen in Figure 2, Creo improves upon the the performance of Thrift for fine-grained data on average of factors 1.16 to 5.23 for C#, Java, and Python clients. Compared to the less mature Thrift C clients, Creo shows improvements of factors 36.84 to 115.69. When comparing the performance of Creo against that of other Java-based tools (illustrated in Figure 3), Creo exhibits performance improvements of on average of factors 5.66 to 388.56, which is attributed to use of more complex serialization techniques and text-resolved data representation formats in other tools.

These tests illustrate the magnitude of serialization overhead for complex serialization techniques, as well as the impact serialization overhead can have on service execution footprint and performance. For example, the JAXB serialization engine used in SOAP tests is unable to process messages of sizes 100 MB in standalone settings, indicating a potential source for load issues when used inside service engines.

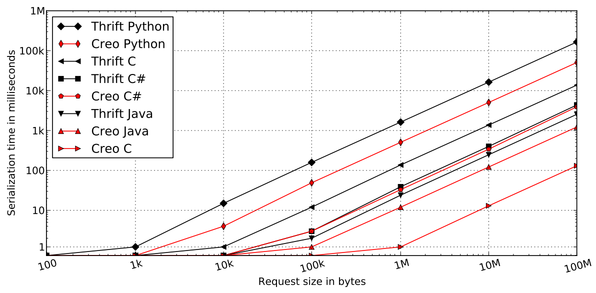
5.3 Transmission Overhead

To evaluate transmission overhead for service communication a simple server component that counts and returns the number of bytes in requests is used. Service invocation makespan is measured on the client side and used to quantify transmission overhead for service invocations with known request payload sizes. Apache Thrift supports transmission of data using three protocols: text-resolved JSON and two binary protocols: TBinaryProtocol and TCompactProtocol, where the former sends data as-is and the latter uses variable-length encoding of integers. The purpose of this encoding scheme; which for example encodes 16-bit integers as 1-3 bytes, 32-bit integers as 1-5 bytes, and 64-bit integer as 1-10 bytes; is to reduce the size of payload and commonly occurring metadata such as the length of strings, arrays, and collections. In tests we primarily use TBinaryProtocol as it is supported in all languages, and evaluate the efficiency of TCompactProtocol in the languages supported (and quantify it against that of Creo and the binary protocol) in separate tests.

For ease of comparison, test results for compact binary representation formats (Creo, Thrift, and JOS) and text-resolved formats (JSON REST and XML

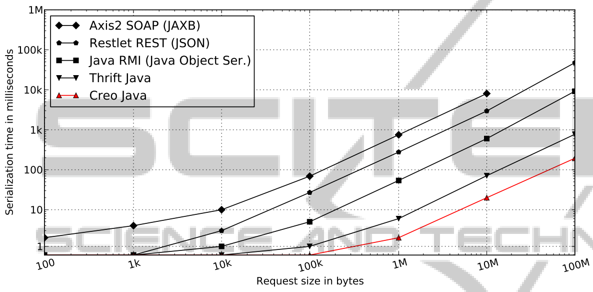


(a) Number-resolved data. On average Creo shows improvements of factors 36.84 (C), 1.23 (C#), 3.51 (Java), and 5.23 (Python) in serialization time.

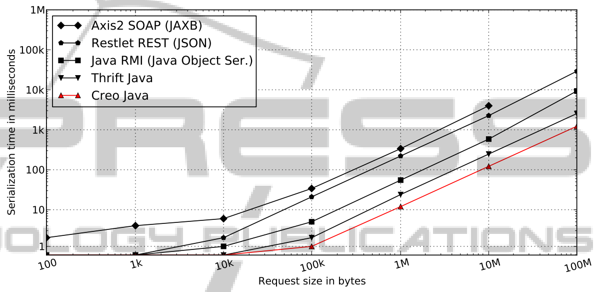


(b) String-resolved data. On average Creo shows improvements of factors 115.69 (C), 1.16 (C#), 2.03 (Java), and 3.24 (Python) in serialization time.

Figure 2: Creo and Thrift serialization time (in milliseconds) for fine-grained messages. Axes logarithmic.

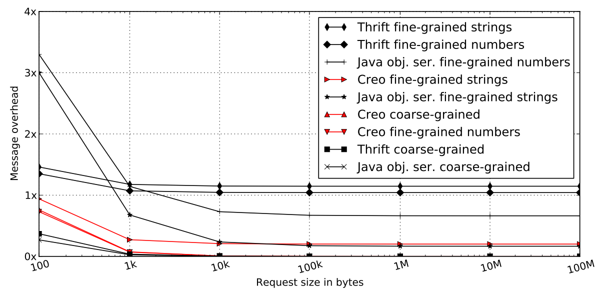


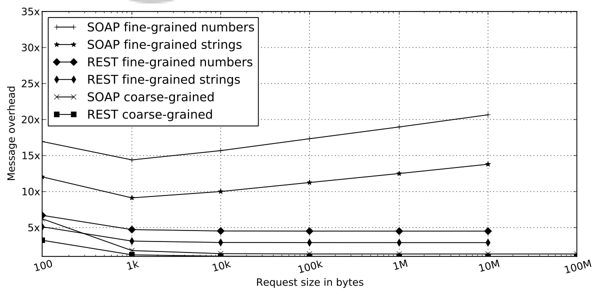
(a) Number-resolved data. On average Creo shows improvements of factors 388.56 (SOAP), 177.18 (REST), and 34.84 (RMI) in serialization time.



(b) String-resolved data. On average Creo shows improvements of factors 30.37 (SOAP), 20.25 (REST), and 5.66 (RMI) in serialization time.

Figure 3: Serialization time (in milliseconds) of Java-based tools for fine-grained messages. Axes logarithmic.





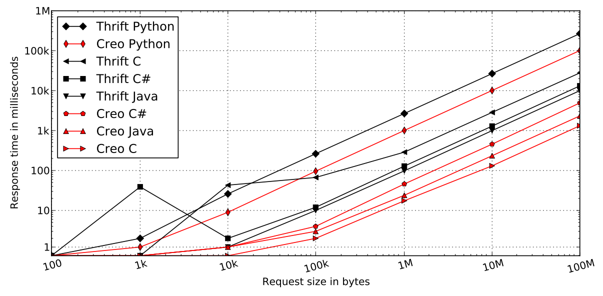
(a) Creo, Thrift, and JOS. (b) REST and SOAP.

Figure 4: Transmission (message size) overhead for service invocation requests. Horizontal axis logarithmic.

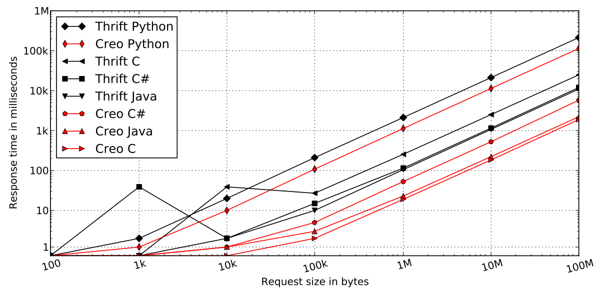
SOAP) are presented separately. As can be seen in Figure 4a, compact coarse-grained (binary) data are represented with little overhead and fine-grained data are represented with overhead within a factor of 2 in size for Creo, Thrift and JOS. As can be seen in Figure 4b, the use of text-resolved representation formats can introduce significant overhead for fine-grained data, ranging in tests up to a factor of 20 for JSON REST and XML SOAP (both of which are unable to process messages larger than 10MB in tests).

5.4 Service Response Time

Having roughly quantified the impact of potential overhead sources for data serialization and transmission, we analyze the communication performance of the evaluated tools in terms of service request response times. Using closed system loop settings (sequential invocations of services deployed in isolated systems), we measure invocation makespan from the client perspective and use it as a measurement of service response time. To verify the transfer of results from sequential tests to (more realistic) parallel invocation scenarios, we also validate results using paral-

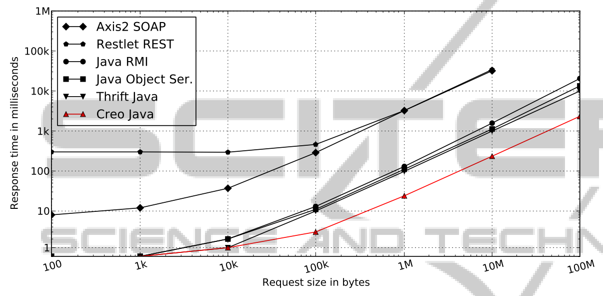


(a) Number-resolved data. On average Creo shows improvements of factors 19.76 (C), 2.77 (C#), 4.21 (Java), and 2.65 (Python) in service response time.

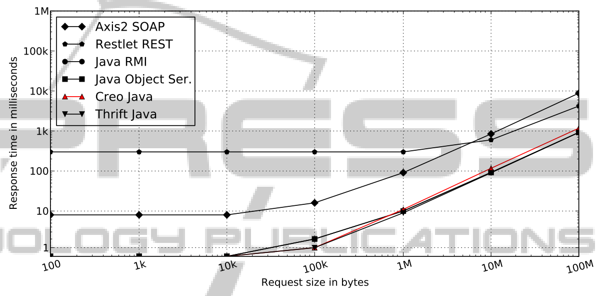


(b) String-resolved data. On average Creo shows improvements of factors 13.42 (C), 2.15 (C#), 4.77 (Java), and 1.87 (Python) in service response time.

Figure 5: Creo and Thrift service response time (in milliseconds) for fine-grained messages. Axes logarithmic.

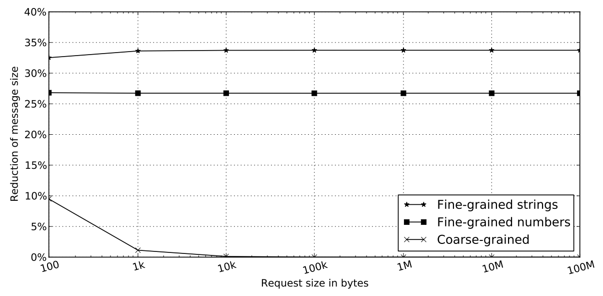


(a) Fine-grained number-resolved data. On average Creo shows improvements of factors 135.69 (SOAP), 140.93 (REST), 6.97 (RMI), and 5.04 (JOS) in service response time.

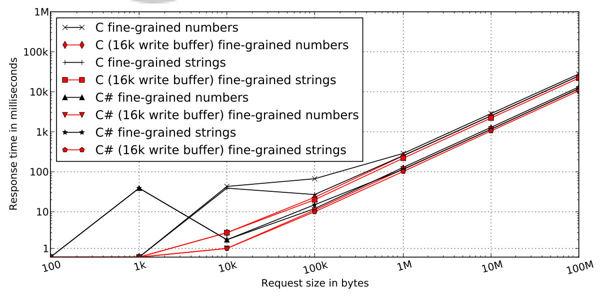


(b) Coarse-grained data. On average Creo shows improvements of factors 7.66 (SOAP), 11.99 (REST), 0.83 (RMI), and 0.83 (JOS) in service response time.

Figure 6: Service response time (in milliseconds) for Java-based tools. Axes logarithmic.



(a) Reduction of service invocation request size for TCompactProtocol compared to TBinaryProtocol.



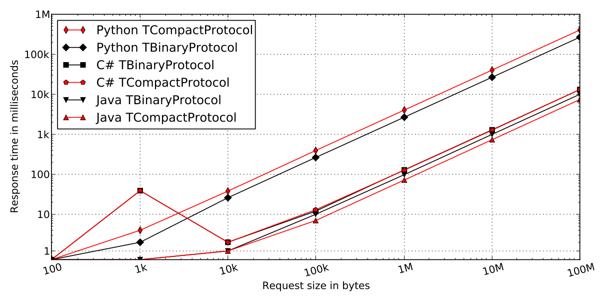
(b) Response time of Thrift's C and C# clients when using 16kB write buffers compared to using the default write buffers.

Figure 7: Transmission overhead in Thrift protocols and buffer alignment issues. Axes logarithmic.

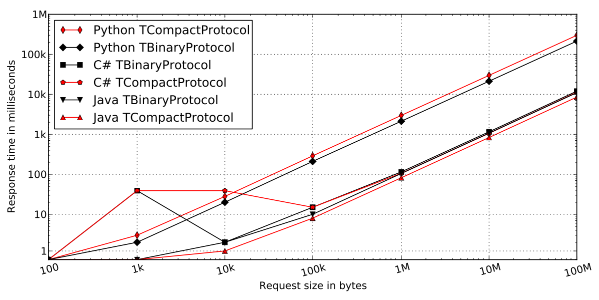
lel invocation tests.

Results from response time tests are visualized in figures 5 and 6. Figure 5 illustrates comparison of the response time of Creo and Thrift services. On average, Creo improves on the response time performance of Thrift for fine-grained data on average of factors 1.87 to 4.77 for C#, Java, and Python clients. Compared to Thrift C clients, Creo shows improvements of factors 13.42 to 19.76. However, for coarse-grained data (unstructured binary data, not illustrated in graphs), Thrift service response times are on average 16% (C), 26% (C#), 27% (Java), and 32% (Python) lower than that of Creo (performance averages calculated for request sizes of 1MB, 10MB, and 100MB). The higher response times of Creo for coarse-grained data are attributed to the use of asynchronous message queues and immutable data structures on the service side, which cause redundant data replications in message transmission.

When comparing the response time of Creo to that of other Java-based tools (illustrated in Figure 6), we note performance improvements of at least factor 4.91 for fine-grained data, and comparative performance for coarse-grained data. As expected from analy-



(a) Number-resolved data. On average TCompactProtocol shows improvements of factors 1.2% (C#), 26% (Java), and -51% (Python) in service response time.



(b) String-resolved data. On average TCompactProtocol shows improvements of factors 7.4% (C#), 22% (Java), and -40% (Python) in service response time.

Figure 8: Response time performance of Thrift protocols. Axes logarithmic.

sis of serialization and transmission overhead, REST and SOAP web services exhibit response time performance degradations from the use of text-based representation formats and associated serializations.

5.5 Thrift Protocols

As mentioned, we use Thrift's TBinaryProtocol in tests as it is supported in all client languages. However, for selected languages, Thrift also supports the TCompactProtocol that in theory provides more efficient representation of data. To ensure fair comparison in tests, we here evaluate the use of this variable-length encoding scheme protocol. As can be seen in Figure 7a, Thrift's TCompactProtocol reduces Thrift transmission overhead of ca 27% (number-resolved data) and 34% (string-resolved data) in tests using fine-grained data. The greater reduction for string-resolved data stems from all test data blocks containing short strings (4 or 10 characters), causing string lengths to be serializable in a single byte. The variable-length encoding scheme has little effect on unstructured (coarse-grained) binary data, but shows an improvement for small messages as the protocol contains less metadata.

In tests, we note oscillations in the performance of Thrift's C and C# clients for data sizes of 1kB and 10kB (see Figure 5). After analysis we speculate that these effects arise due to buffer (size) alignment issues in tests. To investigate this, we evaluate the performance of the same clients with altered buffer sizes, and note (as illustrated in Figure 7b) that the effects can be alleviated using larger (16kB) message transmission buffers.

Finally we evaluate the service response time of Thrift's two binary protocols to investigate the potential impact of Thrift's variable-length encoding scheme on our tests. As illustrated in Figure 8, the TCompactProtocol results in response time improvements of 1.2% to 26% for C and C# clients, and performance degradations of 40% to 51% for Python clients. From these measurements we conclude that use of the TCompactProtocol would not significantly impact the findings of the evaluation.

6 CONCLUSIONS

In this work we investigate an approach to service-based software development and present a toolkit for reduction of complexity in service development and distributed component integration. The architecture of the toolkit is designed to be modular and extensible, and places focus on transparency and reduction of complexity. To reduce learning requirements, the toolkit employs a service description language based on the syntax and type interpretations of the well-known Java language. The service description language defines a set of primitive types and mechanisms for aggregation of types in arrays and structs.

The toolkit supports generation of code for construction of Java-based services as well as service clients in Java, C, C#, and Python. The toolkit uses the same code generation pattern for all languages, which defines immutable types that directly wrap the aggregation patterns defined in service descriptions. For transparency, the service communication protocol serializes data in the order and types defined in the service description language. A performance evaluation quantifying toolkit performance (in terms of overhead and response time) against Java Object Serialization, Java RMI, SOAP web services, REST web services, and Apache Thrift is presented. Toolkit performance is found to be comparable to or improve upon the performance of the alternative techniques.

ACKNOWLEDGEMENTS

The authors acknowledge Mikael Öhman, Sebastian Gröhn, and Anders Häggström for work related to the project. This work is done in collaboration with the High Performance Computing Center North (HPC2N) and is funded by the Swedish Government's strategic research project eSSENCE and the Swedish Research Council (VR) under contract number C0590801 for the project Cloud Control.

REFERENCES

Apache (2005). Apache Web Services Project - Axis2, http://ws.apache.org/axis2, February 2014.

Apache (2009). Apache Avro, http://avro.apache.org/, February 2014.

Apache (2010). Apache Thrift, http://thrift.apache.org/, February 2014.

Armbrust, M., Fox, A., Griffith, R., Joseph, A., Katz, R., Konwinski, A., Lee, G., Patterson, D., Rabkin, A., Stoica, I., et al. (2010). A view of cloud computing. *Communications of the ACM*, 53(4):50–58.

Berl, A., Gelenbe, E., Di Girolamo, M., Giuliani, G., De Meer, H., Dang, M. Q., and Pentikousis, K. (2010). Energy-efficient cloud computing. *The Computer Journal*, 53(7):1045–1051.

Birrell, A. D. and Nelson, B. J. (1984). Implementing remote procedure calls. *ACM Transactions on Computer Systems (TOCS)*, 2(1):39–59.

Fielding, R. T. (2000). *Architectural styles and the design of network-based software architectures*. PhD thesis, University of California.

Furuhashi, S. (2011). MessagePack, https://github.com/ msgpack/msgpack/blob/master/spec.md, February 2014.

Google (2008). https://developers.google.com/protocol-buffers/, February 2014.

Govindaraju, M., Slominski, A., Chiu, K., Liu, P., Van Engelen, R., and Lewis, M. J. (2004). Toward characterizing the performance of soap toolkits. In *Grid Computing, 2004. Proceedings. Fifth IEEE/ACM International Workshop on*, pages 365–372. IEEE.

Jackson (2009). https://github.com/FasterXML/jackson, February 2014.

Kephart, J. O. and Chess, D. M. (2003). The Vision of Autonomic Computing. *Computer*, 36:41–50.

Kryo (2009). https://github.com/EsotericSoftware/kryo, February 2014.

Mackey, A. (2010). Windows communication foundation. In *Introducing. NET 4.0*, pages 159–173. Springer.

MongoDB Inc. (2007). BSON, http://http://bsonspec.org, February 2014.

Oracle (2005). Java Object Serialization, http://docs.oracle.com/javase/7/docs/platform/ serialization/spec/serialTOC.html, February 2014.

Östberg, P.-O. and Elmroth, E. (2011). Increasing Flexibility and Abstracting Complexity in Service-Based Grid and Cloud Software. In F. Leymann, I. I., van Sinderen, M., and Shishkov, B., editors, *Proceedings of CLOSER 2011 - International Conference on Cloud Computing and Services Science*, pages 240–249. SciTePress.

Östberg, P.-O., Espling, D., and Elmroth, E. (2013). Decentralized scalable fairshare scheduling. *Future Generation Computer Systems - The International Journal of Grid Computing and eScience*, 29:130–143.

Östberg, P.-O., Hellander, A., Drawert, B., Elmroth, E., Holmgren, S., and Petzold, L. (2012). Reducing complexity in management of escience computations. In *Proceedings of CCGrid 2012 - The 12th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing*, pages 845–852.

Pautasso, C., Zimmermann, O., and Leymann, F. (2008). Restful web services vs. big web services: making the right architectural decision. In *Proceedings of the 17th international conference on World Wide Web*, pages 805–814. ACM.

Restlet (2013). Restlet Framework, http://restlet.org, February 2014.

Slee, M., Agarwal, A., and Kwiatkowski, M. (2007). Thrift: Scalable cross-language services implementation. *Facebook White Paper*, 5.

Van Deursen, A., Klint, P., and Visser, J. (2000). Domain-specific languages: An annotated bibliography. *Sigplan Notices*, 35(6):26–36.

Vinoski, S. (1993). Distributed object computing with corba. *C++ Report*, 5(6):32–38.

Walker, E. (2009). The real cost of a cpu hour. *Computer*, 42(4):35–41.

Wollrath, A., Riggs, R., and Waldo, J. (1996). A distributed object model for the java system. *Computing Systems*, 9:265–290.