Bringing Middleware to Everyday Programmers with Ballerina

Sanjiva Weerawarana[1], Chathura Ekanayake[1], Srinath Perera[1(✉)], and Frank Leymann[2]

[1] WSO2, Colombo, Sri Lanka
{sanjiva,chathura,srinath}@wso2.com
[2] IAAS, University of Stuttgart, Stuttgart, Germany
frank.leymann@iaas.uni-stuttgart.de

Abstract. Ballerina is a new language for solving integration problems. It is based on insights and best practices derived from languages like BPEL, BPMN, Go, and Java, but also cloud infrastructure systems like Kubernetes. Integration problems were traditionally addressed by dedicated middleware systems such as enterprise service buses, workflow systems and message brokers. However, such systems lack agility required by current integration scenarios, especially for cloud based deployments. This paper discusses how Ballerina solves this problem by bringing integration features into a general purpose programming language.

Keywords: Flow languages · Middleware · Integration technology

1 Introduction

Integration technologies connect various services, APIs and other resources, resulting in meaningful business activities. Since those business activities mirror the real world and interact with the real world, they need to behave accordingly providing security, interruptibility, recoverability, or transactions, respectively.

Often, integration developments are done based on general purpose languages. Those general purpose languages themselves lack the behavior required for integrations mentioned before. Such behavior is provided through middleware. Among examples of those are workflow management systems [1], transaction managers, enterprise service buses, identity gateways, API gateways, and application servers.

However, this behavior is added as an afterthought as middleware. Hence, the corresponding features lack tighter integration with general purpose languages. Therefore, building integration solutions has been a counter-intuitive, complex, and error prone endeavor.

Ballerina[1] is a general purpose, transactional and strongly typed programming language with both, textual and graphical syntaxes. It has the aforementioned behavior natively built in and specialized for the integration domain.

[1] https://ballerina.io.

M. Weske et al. (Eds.): BPM 2018, LNCS 11080, pp. 12–27, 2018.
https://doi.org/10.1007/978-3-319-98648-7_2

It brings fundamental concepts, ideas and tools of distributed systems directly into the language and offers a type-safe, parallel environment to implement such applications. The Ballerina environment has first class support for distributed transactions, reliable messaging, stream processing, workflows and container management.

Furthermore, Ballerina offers simple constructs to control parallel executions. It has a type system designed to simplify the development of distributed applications, for example by providing common message formats such as JSON and XML as native data types. It also includes a distributed security architecture to make it easier to write applications that are secure by design.

Ballerina's support of parallelism, interruptibility, recoverability, and transactions reveals its suitability as basis for a workflow management system. Key workflow patterns [2] especially from the control flow, data flow, and exception handling category can be mapped to Ballerina language elements. For example, parallel execution may be controlled by fork and join statements (see Sect. 5.1), or grouping of steps into compensation-based transactions (Sect. 4.3) as well as ACID-based transactions (Sect. 4.2) are immediate features of the language. It is obvious how many BPMN language elements can be mapped to Ballerina in a straightforward manner. In addition, Ballerina goes beyond that by providing immediate support of resiliency (Sect. 4.5) or security features (Sect. 4.1), for example, to workflows.

This paper will look at how Ballerina makes workflow and other middleware behavior an inherent aspect of a programming language, and how Ballerina helps bringing non-functional properties to everyday programmers.

1.1 History of Integration Technology

Most organizations depend on complex IT infrastructure as they are critical to organizations' successful operations. These infrastructures manage resources, employees, customers, business partners, data related to them, and interactions between them. Such IT infrastructures are called "Enterprise Systems".

Enterprise systems are comprised of many components that are often produced by many different vendors. Single business activities need to use several of these components, i.e. these components need to be integrated. Similarly, since these organizations do business with others, these enterprise systems themselves need to be integrated, i.e. they need to work with other organizations' enterprise systems. System integration connects those different, (even geographically) distributed components into a single system supporting an organization's business. We will call "system integration" just "integration" henceforth.

There are several schools of thoughts for building "enterprise systems" architectures. The leading two are Service Oriented Architecture (SOA) [3] and Resource Oriented Architecture (ROA) [4]. Both these approaches depend on interfaces exposed over the network, which we will call services. Services are defined as "a discrete unit of functionality that can be accessed remotely and acted upon and updated independently, such as an act of retrieving a credit card statement online".

Services that can create a ROA design and built using REST principles [4] are called RESTful services. Such service is called a resource. Each part of an enterprise architecture exposes its capabilities as services (in SOA) or resources (in ROA). Enterprise integration realize the functions of the organization by connecting and orchestrating those services, resources, and different data sources.

Middleware that does system integration is often called "Enterprise Service Bus (ESB)" [5]. ESB is used for designing and implementing communication between mutually interacting software applications (e.g. WSO2 Enterprise Service Bus [6]. When authoring integrations, users need to specify integration logic. Earlier ESBs have used XML (WSO2 ESB [6] and Java based Domain Specific Languages (e.g. CAMEL [7])) for this purpose.

Most programming languages, such as Java, C, C#, Java Script, are based on textual syntax structure. Visual programming [8], where users construct the program visually by composing constructs selected from a palette, is an alternative to textual syntax. ESBs often support visual programming using a data flow model variant [9]. Among examples are WSO2 ESB's Studio [6], Mule's Anypoint Editor [10], and Boomi Editor [11].

Earlier architectures supporting integrations were centralized in the sense that ESB(s) host all the integration logic. Later, with the emergence of microservices architecture, focus was given to develop integration logic as independent entities, where each integration flow can be deployed in its own process. Container technologies such as Docker have made this idea practical. Integration logic used within microservices architecture is called micro-integrations. We use the word "integration" to encompass both centralized integration as well as "micro-integrations".

Some of these services are published and consumed within a single trust domain (e.g. a single organization). However, services may be published, shared and consumed beyond a single trust domain. We call them remote APIs, in this context often abbreviated to just APIs. Middleware that manage APIs is called "API Manager" [12]. API management lets the organization have a greater control over creating and publishing APIs, enforcing their usage policies, controlling access, nurturing the subscriber community, and collecting and analyzing usage statistics. Integration logic that is available as an API is called an API composition.

Among related work, software development using visual interfaces is discussed in [8]. UML [13] is one of those visual languages. Sequence Diagrams (SDs) that capture the flow of executions are also a part of UML. Sequence diagrams have been used to model service based applications [14]. Koch et al. [15] describes the use of UML for web engineering. Ermagan et al. [16] describe a new UML profile for modeling services. Furthermore, sequence diagrams have been used for modeling tests cases [17] and High level Petri Nets [18].

1.2 The Importance of Non-functional Behaviors

Primary focus of a programming language is the grammar to specify how various elements of the language can be combined into a valid program. Almost

all programming languages provide constructs for data flow (e.g. variable definitions and variable assignments), arithmetic operations, and control flow (e.g. conditional branching and loop). A language also typically supports exception handling, controlling parallel executions and input/output processing.

However, non-functional properties such as availability, resiliency, data integrity, security, observability, recoverability and reliability become important factors when programming in distributed systems environments. In a distributed environment, a single use case may involve many services hosted on different servers. It is possible that one or more of those services become unavailable in certain time periods. In such situations, it should be possible to identify failed services and reroute requests to alternative services if possible or handle the error situation gracefully without causing major disruption to the overall operation. Furthermore, if a critical operation failed due to some reason, the system should guarantee the integrity of itself as well as all interacting systems. For example, assume that a system is handling travel arrangements for a conference involving multiple external services for airline tickets, conference registration and hotel reservations. If an air ticket could not be reserved on required dates after registering for the conference, it may be required to cancel the conference registration, which in turn should adjust expected participants for relevant tracks and seating requirements. If all these functions are handled by services, a failure at invoking the airline service should trigger cancellation actions in conference registration service and all relevant services invoked by conference registration service.

Another important aspect is the reliable delivery of messages in an environment where any component can fail, and the number of messages produced per unit time by a sending component may not match the number of messages that can be consumed per unit time by its receiving component. In such situations, a system that sends a message should have a guarantee that the target system will receive the message even if that system is not available at that moment.

A more critical non-functional requirement from the workflow perspective is the interruptibility, which ensures to resume work from the last checkpoint after a system failure. Depending on the check pointing behavior, a system can avoid redoing expensive operations and prevent unnecessary compensating actions. As an example, assume that an order handling process invokes an inventory service to allocate goods to be shipped. Further, assume that the inventory service involves some manual tasks and hence takes few days to complete. Now if the order handling process crashes (e.g. due to an unexpected shutdown of the server) after receiving the response from the inventory service, whole state of the process would be lost. If interruptibility is not supported, order process has to be started from the beginning after the server restart, which in turn invokes the inventory service again causing few days of delay. Additionally, some compensation action is required to deallocate previously allocated goods. None of these problems would arise if recoverability is supported by the order processing system, which can create a checkpoint immediately after receiving the response from the inventory service, thus allowing the system to resume from that checkpoint after a failure.

Such non-functional properties are not available in most of the current programming languages as they are not developed for systems integration. In order to overcome limitations of current programming languages, integration developers have to use middleware systems such as workflow engines, enterprise service buses, messages brokers or transaction managers. A typical integration use case may require functionality offered by many of these systems. Furthermore, functionality provided by middleware systems alone is not sufficient to address most integration use cases forcing developers to build solutions by combining code written in programming languages with middleware systems. This often results in complex solutions consisting of disparate set of systems that require diverse skill sets and infrastructure.

2 Language Philosophy

Ballerina aims at simplifying integration development by bringing in key features required for systems integration such as efficient communications over multiple protocols, resiliency, recoverability, and transactions, into a programming language. Therefore, developers can develop integration and workflow style solutions in the same way as any other application. For example, if it is necessary to invoke an HTTP endpoint within a program, a developer may create an endpoint with necessary resiliency parameters such as retry interval, retry count and failure threshold, and invoke the endpoint. Ballerina runtime will handle all necessary details such as invoking the endpoint without blocking OS threads, taking specified actions on endpoint failures and facilitating recovery from runtime failures. Integration features offered by Ballerina are discussed in later sections.

Ballerina can be used to develop many integration solutions. Developers can utilize its service publishing capabilities over multiple protocols to develop microservices. They can combine its service publishing capabilities with message processing and endpoint features to develop micro-integrations. Furthermore, microworkflows can be developed using features such as recoverability, compensating transactions and ability receive multiple inputs. All these solutions result in executable Ballerina programs which can be executed in stand-alone mode without any middleware support.

For many years we observed repeatedly that, when faced with a complex integration problem, various stakeholders often draw a sequence diagram to develop a solution to their problem. Hence, a sequence diagram is a suitable visual representation that has a shared meaning among integration programmers and architects.

Ballerina uses UML sequence diagrams (SDs) to build integration solutions. The Ballerina editor includes a visual and a textual view side by side. Programmers can switch between either view in the middle, apply changes in any of the views, and lossless conversion happens immediately. This enables the user to build the integration logic either using a visual syntax or using textual syntax.

Another key aspect is to use automatic analysis and intelligent predictions to reduce the burden on the developer. Unlike middleware implemented separately

from general purpose languages, Ballerina has access to the whole program and can fully control its execution. Therefore, it can perform automatic analysis effectively and optimize the execution of a program. Among examples are locks that will be automatically acquired and released allowing maximal concurrency, auto tuning thread pools, and taint checking built into the language.

3 Language Elements

A detailed description of Ballerina can be found in the language specification [14]. In this section, we will explore some of its key features.

A typical program includes a service or a main program. Both main program and resource is built with many statements. Each statement may define data types, call functions, define and create workers, and run control statements such as loops and conditions. Furthermore, Ballerina supports annotations to associate configurations or additional behaviors with language constructs.

3.1 Type System

Ballerina type system includes three types of values: (i) simple values such as booleans and integers, (ii) structured values such as maps and arrays and (iii) behavioral values like functions and streams. Ballerina considers JSON and XML as native structured types simplifying the handling of message payloads used in most integration scenarios.

In addition to the above, *union*, *optional* and *any* types are supported. *Union* type allows a variable to contain values of more than one type. This is useful for example for a function to return a string or an error depending on the processing outcomes. *Optional* indicates that a variable can be null. Variables of *any* type can be assigned with values of any type defined in Ballerina type system, making it useful for scenarios where the type of the variable is not known at development type.

3.2 Connectors and Endpoints

Interactions with external APIs and services is a fundamental requirement of integration. Ballerina uses connectors to program interactions with external entities. Such entities can be generic HTTP endpoints, databases or specific services such as Twitter, GMail, etc. A connector can be initialized by providing a configuration required for the corresponding external service. For example, a simple HTTP connector initialization would look like:

```
endpoint http:Client taxiEP {
   url: "http://www.qtaxies.com"
};
```

Once initialized, any action supported by the connector can be invoked by providing appropriate parameters as below:

```
taxiEP -> get("/bookings");
```

The connector concept was introduced in order to differentiate network calls from other function invocations, so that Ballerina developer is aware of associated concerns such as latency, probability of failures and security implications. In fact, Ballerina addresses some of these concerns using resiliency features discussed in Sect. 4.5. In addition to the connectors shipped with Ballerina, it is possible to develop custom connectors for any programmable endpoint and use them in the same way as any other existing connector.

3.3 Error Handling

Programming languages handle errors in two ways. The code can either stop the execution of the normal flow or return the details back into upper levels to handle or recover from the error. Languages like Java can support both the above methods by throwing exceptions. However, if it is recoverable, it is expensive to unwind and do exception processing, and it is considered an anti-pattern to throw exceptions in the normal execution flow.

In such cases, languages like Java can return error details (without throwing exceptions), which must be checked at upper level. However, if upper levels forget to check it, it will lead to errors.

Ballerina handles both above cases by introducing a first-class error concept which can both be returned and thrown. Thrown errors behave just like exceptions and cause the call stack to be unwound until a matching catcher is found. However, the language forces the upper levels to check the return value for errors. It is done by returning a union type of type T|error. The error part has to be handled using a match expression or explicitly declaring that the error is not important at assignment via T|_. Unless this is done, the compiler will complain when the program tries to access the value of an union type.

3.4 Ability to Inject Values into a Running Executions

Once a program instance is started, it may be necessary to get additional inputs after completing certain steps. Such inputs can come from other parallel flows of the same program instance or from external systems. The former case can be implemented by most programming languages using shared variables with appropriate synchronization mechanisms. For example, in Java a thread can wait on an object to be notified by another thread once required data is available. Ballerina facilitates parallel executions based on a construct named *worker* as explained in Sect. 5.1. Accordingly, it provides an inter worker communication method to pass variables among workers as below:

```
function f1() {
   worker w1 {
      string city = "NY";
      city -> w2;
```

```
    }
    worker w2 {
        string location;
        location <- w1;
    }
}
```

In the above example, worker *w2* waits for an input from worker *w1* and once the input is available, assigns it to a variable named location.

The more complex scenario is receiving intermediate inputs from external systems. In this scenario, a process instance is already started and it expects an external input in order to continue. Furthermore, there can be many instances of the same program running at the same time. Therefore, the main challenge here is to identify the correct process instance to deliver the message. A program instance can be identified either using a unique instance identifier or using a combination of variables whose values will be unique to an instance. Such combinations of variables are called correlation variables. The latter approach is more flexible as it does not force external systems to be aware of program instance identifiers. Instead, such systems can just send messages including values for correlation variables, which are, in most cases, business variables such as customer identifiers, order numbers, etc. Many workflow systems use this correlation variable based approach for receiving intermediate inputs [19]. Inspired by this, Ballerina brings a similar concept to programming language level by allowing programs to receive intermediate inputs, where the Ballerina runtime environment correlates incoming messages with relevant program instances based on variables defined in the program flow. The below code fragment shows how a Ballerina program can receive the location of a customer:

```
string customer = "smith";
json correlationVars = {"custId": customer};
queue:Message result = custEP -> receive(correlationVars);
string location = result.getTextMessageContent();
```

Any Ballerina program can send messages to such intermediate reception points by providing values for correlation variables as below:

```
string city = "NY";
queue:Message loc = custEP.createTextMessage(city);
json correlationVars = {"custId": "smith" };
loc.setCorrelationID(correlationVars.toString());
custEP -> send(loc);
```

Thus, it is possible to write Ballerina programs that get messages from any source such as HTTP services, JMS or file systems and trigger other Ballerina programs waiting on intermediate inputs. An advantage of this approach is the flexibility on how external systems can send correlated messages. For example, if an external system can only send JMS messages, it is possible to write a Ballerina program to receive JMS messages and trigger a waiting Ballerina program.

4 Non-functional Properties

4.1 Security

Ballerina guarantees security for sensitive parameters using a compiler level taint checking mechanism. There can be functions and connector actions that take sensitive parameters such as SQL queries or file paths. Assigning untrusted values into those parameters can cause major security vulnerabilities such as SQL injection attacks. In order to prevent such vulnerabilities, Ballerina programmers can mark relevant parameters with the @sensitive annotation as below:

```
function selectData(@sensitive string query, string params)
        returns string { .. }
```

If a tainted variable (i.e. a variable that may contain an untrusted value) is assigned to a parameter marked with *@sensitive* annotation, Ballerina compiler will produce an error. Furthermore, sensitivity of parameters is automatically inferred when calling functions. This can be illustrated using the below sample code:

```
function getAddress(string username) {
   string query = "select address from user" +
                  "where uid = '" + username + "'";
   string address = selectData(query, null);
}
```

In this case, *username* parameter of the *getAddress* function is also considered sensitive by the compiler as its value is used to derive the sensitive query parameter of the invoked *selectData* function. Similarly, functions can mark return values as tainted to indicate that they are not safe to be used as sensitive parameters. This is done using the *@tainted* annotation as below:

```
function readUsername() returns @tainted string { ... }
```

4.2 Distributed ACID Transactions

Being an integration language, Ballerina programs have to interact with many external systems including databases, message brokers and other services, which are often required to be grouped into a single unit of work. Therefore, Ballerina programs have to support distributed transactions involving all participating entities to ensure integrity of the corresponding integration scenarios. Ballerina language provides constructs to mark transaction boundaries to coordinate the joint outcome of invoked endpoints based on a 2PC protocol. Below is an example of a transaction within a Ballerina program:

```
transaction with retries = 0, oncommit = commitFunc, onabort = abortFunc
{
   _ = bankDB -> update("UPDATE ACCOUNT SET BALANCE = (BALANCE - ?)" +
```

```
                "WHERE ID = ?", 1000, 'user1');
    match depositMoney("acc1", 1000) {
        error depositError => {
           abort;
        }
        () => isSuccessful = true;
    }
}
```

In this example, the *update* action of a database connector (i.e. bankDB) is invoked to decrease the amount to be transferred. However, if the *depositMoney* function fails, the transaction is aborted so that Ballerina runtime will rollback the database transaction as well. In addition to coordinating transactions of invoked endpoints, Ballerina programs can participate in distributed transactions as well. If a Ballerina program *B1* invokes another Ballerina program *B2* within a transaction, *B2* automatically participates in the transaction initiated by *B1*. If *B2* also defines a transaction, *B2*'s transaction will not be commited until *B1* is ready to commit. Similarly, if *B2* invokes another Ballerina program *B3*, *B1*'s transaction is propagated to *B3* as well (transaction infection).

4.3 Compensation-Based Transactions

Compensation is another technique of maintaining integrity [20], especially for long running interactions or interactions that do not support ACID semantics. Distributed ACID transactions mentioned above are inherently subject to blocking. This is not be suitable for tasks that span longer time periods (more than several seconds). Instead, compensation-based mechanisms allow each participating entity to commit work immediately. In addition, each such entity has to provide a corresponding compensation action, which will be triggered if the overall task fails. Such compensation mechanisms are implemented in many workflow systems and Ballerina language introduces this concept at programming language level. In workflow languages such as BPMN, compensation actions can be associated with BPMN activities using compensation boundary events. However, programming language statements are too fine grained as compensable units. Therefore, a grouping construct named *scope* is introduced (like in BPEL), so that compensation actions can be associated with a scope containing multiple statements. Scopes can be nested. A construct named *compensate* is introduced to trigger compensations of completed scopes. It can be invoked with or without a scope name. If a scope name is given, only the named scope and its child scopes are compensated. If a scope name is not given, all completed scopes within the current scope are compensated. Compensation actions are triggered in the reverse of the completion order. Following is a Ballerina code snippet with compensations:

```
scope s1 {
   scope s2 {
      ...
```

```
   } compensation(var2, var3) { ... }
   scope s3 {
        result = ...
    } compensation(var4) { ... }
    if (result == -1) {
       compensate();
    }
} compensation (var1) { ... }
```

In the above example, if the compensation is triggered, compensation actions of scopes *s3* and *s2* are invoked in that order. However, the compensation actions of scope *s1* will not be invoked as *s1* is not completed at the time of compensation.

4.4 Interruptibility

Compiled Ballerina programs run on Ballerina Virtual Machine (BVM) as discussed in Sect. 5. BVM supports interruptibility by persisting the state of running programs. A persisted state is a special kind of checkpoint. If a connector developer (not Ballerina programmer) has indicated that a certain connector action should not be repeated after a recovery, BVM makes a checkpoint whenever a program completes that action. Such actions can include invocations of external services or database operations. For example, if a service is invoked to reserve a ticket, it should not be re-invoked when the invoking program is resumed after a failure. Similarly, if a connector developer has indicated that an action can take long time to complete, BVM makes a checkpoint before a program invokes that action. An example of such action is the waiting for a reception of an intermediate input as mentioned in Sect. 3.4. As such operations can take long time periods, there is more probability of failure at those points. Therefore, checkpointing before starting such operations allows BVM to resume programs from those points in case of a failure. Furthermore, Ballerina has a language element named checkpoint in order to allow programmers to define checkpoints anywhere within a program. If the BVM stops due to any reason (e.g. server crash) and is subsequently restarted, it will resume all program executions from the last available checkpoint.

When a Ballerina program is invoked as an HTTP resource, the program can send a reply back to the client using an HTTP response message. However, if the BVM is restarted, the underlying TCP connection with the client will have been terminated and BVM will not get a connection to send the HTTP response. In this scenario, a client can resend the original request, which will be correlated by the BVM with the corresponding program instance. Then once the program reaches the replying point, it will reply using the new correlated connection. If the program has reached the replying point before receiving the correlating request, the program state will be saved until such request is received. Thus, Ballerina supports request-response behavior for long running flows even after system failures.

4.5 Resiliency

Distributed environments that Ballerina programs are expected run may contain unreliable networks, failure prone servers, overloaded systems, etc. In order to facilitate development of robust programs for such environments, Ballerina provides a set of resiliency features, namely circuit breaking, failing over, load balancing and retrying with timeouts. Circuit breaking allows programmers to associate suspension policies with connectors so that connectors stop sending messages to unresponsive endpoints if suspension criteria is met. For example, an HTTP connector can be configured to suspend further requests for 5 min if more than 5% of requests fail within 30 s.

Similarly, failover configurations can be associated with connectors to select alternative endpoints if one endpoint fails. Load balancing configuration specifies a set of endpoints and a load balancing algorithm, so that requests are distributed among specified endpoints according to the given algorithm. This is useful to avoid overloading backend systems, especially where a dedicated load balancer is not available. Finally, a retry configurations can be defined for connectors to force the connector to retry sending the request in case of a failure.

5 Architectural Aspects

Ballerina is a compiled language. Ballerina compiler takes Ballerina programs as input and generates intermediate code. This intermediate code can be executed in Ballerina Virtual Machine (BVM). The Ballerina compiler performs syntax checks and transforms programs into an intermediate format containing instructions understood by the BVM. The BVM performs the fetch-, decode-, execute-cycle acting as a CPU for Ballerina intermediate code. In addition to executing instructions, the BVM performs tasks such as listening for incoming messages, concurrency control, exception handling and transaction management.

5.1 Thread Model

Ballerina is a parallel language and natively support parallel executions. Each execution of a resource or a main program has implicit workers. However, workers may be created explicitly, they can safely talk to each other, synchronize data, and support complex scenarios such as fork and join based on corresponding language syntax.

A worker can be considered as a sequence of Ballerina instructions with a storage to store variables, input arguments and return values used within those instructions. BVM can execute a worker by assigning it to an OS thread. A worker that needs to be executed synchronously is run in the same thread as its invoker (e.g. synchronous function calls). Asynchronously executed workers are assigned to new threads taken from a thread pool. BVM executes a worker in its assigned OS thread until a blocking instruction is reached (e.g. sleep or connector invocation). At this point, BVM saves the context of the worker in an

appropriate callback function and releases the assigned OS thread. Therefore, the OS thread previously assigned for the blocked worker will become available to run an unblocked worker. Once the blocking action returns and its callback is invoked, the callback function gets the saved worker context and lets BVM to execute it by assigning an OS thread.

Each Ballerina function can have one or more workers. Ballerina programmers can define workers explicitly within a function as below:

```
function f1() {
   worker w1 { ... }
   worker w2 { ... }
}
```

In this case, workers *w1* and *w2* will be executed in parallel. If workers are not defined within a function, a default worker is associated with it by the BVM. In addition to the workers associated with functions, Ballerina provides fork/join constructs to trigger parallel flows as below:

```
fork {
   worker w1 { ... }
   worker w2 { ... }
} join (all) { ... }
```

Furthermore, it is also possible to start any function asynchronously by using the start keyword as below:

```
future<int> result = start f2();
...
int value = await result;
```

In this case, the future statement immediately returns without waiting for *f2* to complete. Then the Ballerina program can call await at any point later in the program to wait for the function to complete and get the result.

A common problem in parallel programs is to handle shared data safely. Ballerina supports this via a lock statement. Ballerina goes beyond languages like Java by automatically analyzing the locked data structures, figuring out minimal shared scope, and then locking at that level to allow maximal concurrency.

5.2 Non-blocking I/O

Ballerina supports non-blocking I/O without any additional overhead for programmers. From a programmer's perspective, I/O calls work in a blocking manner, so that the statement immediately after the I/O call is executed only after the I/O call returns with a response. An invoking program can access the response immediately as shown below:

```
var result = clientEndpoint -> get("/get?test=123");
io:println(result);
```

However, according to the thread model discussed in Sect. 5.1, BVM releases the underlying OS thread whenever an I/O call is made and stores the program state in a memory structure along with next instruction pointer. When the result of the I/O call is available, BVM allocates a new thread from its thread pool to continue the saved program state. Therefore, Ballerina programmer sees it as a blocking I/O call although OS threads are not blocked.

5.3 Observability

Observability is a measure of how well internal states of a system can be inferred. Monitoring, logging, and distributed tracing can be used to reveal the internal state of a system to provide observability. Ballerina becomes observable by exposing itself via these three methods to various external systems allowing them to monitor metrics such as request count and response time statistics, analyze logs, and perform distributed tracing. It follows open standards when exposing observability information so that any compatible third party tool can be used to collect, analyze and visualize information.

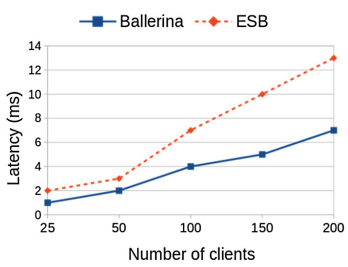
6 Measurements

Performance is a critical factor for integration software, as a typical deployment may have to connect with many external systems and process thousands of requests per second. These requests may be sent by hundreds of different clients. Two commonly used measurements for evaluating performance are latency and throughput. Latency of a request is the round trip time between sending a request to a system and receiving a response. Throughput is the number of requests that can be processed by a system in a unit time.

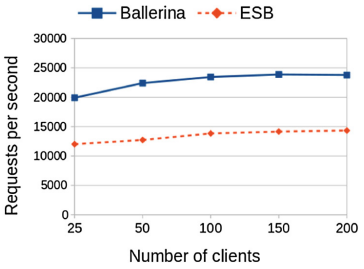
We compared these two measurements of Ballerina with those of WSO2 ESB for different concurrency levels. A basic integration scenario of receiving a message from a client over HTTP, sending it to a backend system, receiving the response from the backend and sending the response back to the client is considered for these tests. A Message of size 1kB is used and zero backend processing time is simulated (i.e. backend responds immediately). Tests are conducted on Intel Core i7-3520M 2.90GHz 4 machines with 8 GB RAM and JVM heap size of 2 GB is allocated. Ballerina/WSO2 ESB, client (JMeter[2]) and the backend (WSO2 MSF4J[3]) are run on three separate machines. Results for latency and throughput are shown in Fig. 1(a) and (b) respectively:

According to the results, Ballerina outperforms ESB at all concurrency levels. Ballerina can process a request in 5 ms for 200 concurrent clients whereas ESB takes 10 ms for the same scenario. Performance difference is also significant for throughput, where Ballerina shows around 24000 transactions per second (TPS) for 200 concurrent clients while ESB shows only around 14000 TPS.

[2] https://jmeter.apache.org.
[3] https://wso2.com/products/microservices-framework-for-java.



(a) Variation of latency with number of concurrent clients



(b) Variation of throughput with number of concurrent clients

Fig. 1. Comparison of Ballerina with ESB

These results indicate that Ballerina can process requests faster and serve more requests concurrently without considerably degrading performance, which is a desirable property for integration systems.

7 Conclusion and Outlook

The main objective of the Ballerina project is to create a programming language for integration. Therefore, in addition to providing general purpose programming constructs, Ballerina has built-in support for a broad set of integration features. Such features include efficient service invocations, listening for incoming connections, transactions, support for multiple protocols and simplified database access. By recognizing the possible long running nature of certain integrations, some features of workflow systems were absorbed into Ballerina. As a result, persistence based interruptibility is introduced to support recovery from unexpected failures during long-running workflows. Furthermore, compensation is introduced to support long-running transactions. Then, the ability to receive intermediate inputs with correlations were implemented, so that a single Ballerina program can receive messages from any number of channels. Combining all these features, Ballerina can be used as a programming language for programming short running integrations as well as long-running workflows.

As future work, we are planning to improve the current Ballerina workflow implementation to production ready state. In addition, once Ballerina constructs equivalent to other critical BPMN constructs are introduced, a (partial) mapping from BPMN to Ballerina can be performed. By extending this idea, it is possible to develop a BPMN editor, which generates Ballerina code that can run in the BVM.

From the runtime perspective, we are planning to implement the BVM using the LLVM infrastructure in order to increase performance.

References

1. Leymann, F., Roller, D.: Production Workflow: Concepts and Techniques. Prentice Hall PTR, Upper Saddle River (2000)
2. Workflow patterns. http://www.workflowpatterns.com/. Accessed 10 Jun 2018
3. Erl, T.: Service-Oriented Architecture: Concepts, Technology, and Design. Pearson Education India, Noida (2005)
4. Duggan, D.: Enterprise Software Architecture and Design: Entities, Services, and Resources, vol. 10. Wiley, Hoboken (2012)
5. Schmidt, M.-T., Hutchison, B., Lambros, P., Phippen, R.: The enterprise service bus: making service-oriented architecture real. IBM Syst. J. **44**(4), 781–797 (2005)
6. WSO2 Enterprise Service Bus. http://wso2.com/products/enterprise-service-bus/. Accessed 30 May 2018
7. Apache camel. http://camel.apache.org/. Accessed 30 May 2018
8. Sundararajan, P., et al.: Software development using visual interfaces. US Patent 7,793,258, 7 September 2010
9. Hils, D.D.: Visual languages and computing survey: data flow visual programming languages. J. Vis. Lang. Comput. **3**(1), 69–101 (1992)
10. Mule anypoint studio. https://www.mulesoft.com/lp/dl/studio. Accessed 30 May 2018
11. Dell boomi platform. https://boomi.com/platform/integrate/. Accessed 30 May 2018
12. API management. https://en.wikipedia.org/wiki/API_management. Accessed 30 May 2018
13. O.UML, Unified Modeling Language. Object Management Group (2001)
14. Ballerina language specification, v0.970, working draft. https://ballerina.io/res/Ballerina-Language-Specification-WD-2018-05-01.pdf. Accessed 30 May 2018
15. Koch, N., Kraus, A.: The expressive power of UML-based web engineering. In: Second International Workshop on Web-oriented Software Technology (IWWOST 2002), vol. 16. CYTED (2002)
16. Ermagan, V., Krüger, I.H.: A UML2 profile for service modeling. In: Engels, G., Opdyke, B., Schmidt, D.C., Weil, F. (eds.) MODELS 2007. LNCS, vol. 4735, pp. 360–374. Springer, Heidelberg (2007). https://doi.org/10.1007/978-3-540-75209-7_25
17. Sarma, M., Kundu, D., Mall, R.: Automatic test case generation from UML sequence diagram. In: 2007 International Conference on Advanced Computing and Communications, ADCOM 2007, pp. 60–67. IEEE (2007)
18. Alhroob, A., Dahal, K., Hossain, A.: Transforming UML sequence diagram to high level petri net. In: 2010 2nd International Conference on Software Technology and Engineering (ICSTE), vol. 1, pp. V1–260. IEEE (2010)
19. Görlach, K., Leymann, F., Claus, V.: Unified execution of service compositions. In: Proceedings of the 6th IEEE International (2013)
20. Leymann, F.: Supporting business transactions via partial backward recovery in workflow management systems. In: Lausen, G. (ed.) Datenbanksysteme in Büro, Technik und Wissenschaft, pp. 51–70. Springer, Heidelberg (1995). https://doi.org/10.1007/978-3-642-79646-3_4