Decentralized Orchestration of Composite Web Services

Walter Binder, Ion Constantinescu, Boi Faltings
Ecole Polytechnique Fédérale de Lausanne (EPFL)
Artificial Intelligence Laboratory
CH–1015 Lausanne, Switzerland
Email: firstname.lastname@epfl.ch

Abstract

Traditional, centralized orchestration of composite web services often leads to inefficient routing of messages. To solve this problem, we present a novel scheme to execute composite web services in a fully decentralized way. We introduce service invocation triggers, a lightweight infrastructure that routes messages directly from the producing service to the consuming one, enabling fully decentralized orchestration. An evaluation confirms that decentralized orchestration can significantly reduce the network traffic when compared with centralized orchestration.

Keywords: *Composite web services, decentralized orchestration, workflows*

1. Introduction

Service-oriented computing, a new approach to software development, enables the construction of distributed applications by integrating services that are available over the web [10]. The building blocks of such applications are web services[1] that are accessed using standard protocols. The composition of individual web services into an added-value, composite web service is usually represented as a workflow.

In previous work [7, 5] we presented a flexible and efficient framework for the fully automated generation of composite web services based on a given service request and a potentially large-scale repository of web service advertisements. In this paper we complement our service composition infrastructure with a mechanism for the efficient, distributed execution of composite web services represented as workflows.

Even though a workflow may invoke services distributed over multiple servers, the orchestration of the workflow is typically centralized. E.g., BPWS4J [4] acts as centralized coordinator for all interactions among the individual services within a workflow. While this approach gives complete control over the workflow orchestration to a single entity (which may monitor the progress), it often leads to inefficient communication, as all intermediary results are transmitted to the central workflow orchestration site, which may become a bottleneck. This is particularly problematic, if a workflow is executed on a mobile device with limited or expensive network connection.

The contribution of this paper is a novel scheme of fully decentralized workflow orchestration. We introduce *service invocation triggers*, in short *triggers*, which act as proxies for individual service invocations. Triggers collect the required input data before they invoke the service, i.e., triggers are also buffers. Moreover, they forward service outputs to exactly those sites where they are actually needed, supporting multicast. In order to make use of triggers, workflows are decomposed into sequential fragments, which contain neither loops nor conditionals, and the data dependencies are encoded within the triggers. Once the trigger of the first service in a workflow has received all input data, the execution of that service is started and the outputs are forwarded to the triggers of subsequent services. Consequently, the workflow is executed in a fully decentralized way, the data is transmitted directly from the producer to all consumers.

For the discussion in this paper, a simplified formalism to describe services is sufficient. We describe a service by a set of input and a set of output parameters. Each input (resp. output) parameter has an associated name that is unique with the set of input (resp. output) parameters. We assume the workflow of a composite service to be consistent with the specifications of the individual services. Hence, we do not consider the type of parameters, as we presume that whenever a service receives an input for a particular parameter, the actual type of the passed value corresponds to the formal type of that service parameter.

We assume that services are invoked by remote procedure calls (RPC), such as SOAP RPC [14]. The values

[1] We use the terms *web service* and *service* interchangeably.

for the input parameters are provided in a request message, while the values for the output parameters are returned in a response message. Asynchronous (one-way) calls can be easily mapped to RPC, which is actually the case for SOAP over HTTP. Our triggers are designed to be transparent to services, i.e., services do not need to know whether they are invoked directly by a client or by a trigger. Therefore, our framework can be deployed without changing existing services.

The rest of this paper is organized as follows: Section 2 introduces the concept of triggers, while Section 3 presents a simple API to create and manipulate triggers. Section 4 explains how workflows are mapped to triggers in order to execute them in a fully decentralized way, which is illustrated with an example. In Section 5 we consider the handling of failures during workflow execution. Section 6 treats security aspects related to the use of triggers. Section 7 presents an evaluation confirming that decentralized orchestration allows to significantly reduce network traffic in comparison with centralized orchestration. Finally, Section 8 discussed related work and Section 9 concludes this paper.

2. Service Invocation Triggers

In this section we give an overview of service invocation triggers, the main abstraction in our framework for efficient, decentralized workflow orchestration. A trigger corresponds to one invocation of a service. Hence, it can be considered a specialized proxy for a single service invocation. A trigger plays four different roles:

1. It collects the input parameter values for one service invocation.

2. It acts as a message buffer, as each input parameter value may be transmitted by a distinct sender at a different time.

3. It triggers the service invocation, after a value has been received for all required input parameters (synchronization). In order to invoke the service, the trigger assembles a RPC request message.

4. It defines the routing for each output parameter value of the service. As each output parameter value may be routed differently, the trigger may have to split the RPC response message returned by the service upon invocation. Each output parameter value may be routed to multiple different triggers, i.e., triggers support multicasting.

With the aid of triggers it is possible to distribute the knowledge concerning the data dependencies of the services

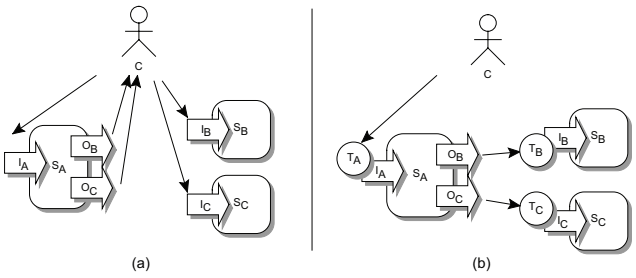

Figure 1. (a) Centralized orchestration versus (b) decentralized orchestration using triggers.

within a workflow. The main difference between the workflow and the corresponding triggers is that the triggers are distributed and attached to services. Each trigger defines which service to invoke. The trigger waits until all required input parameter values are available before it fires (i.e., triggers the service invocation). Moreover, each trigger encapsulates workflow-specific knowledge where the output parameter values of the service invocation are needed. As the trigger acts as a proxy for the service, it receives the output parameter values and forwards them to other triggers according to its routing information.

The following example (see Figure 1) shows how triggers can help to optimize the data transmission between services: Consider a service S_A which requires a single input parameter I_A and produces two output parameters O_B and O_C, while the services S_B and S_C both require a single input parameter, I_B resp. I_C. Further, assume a workflow fragment that requires an invocation of S_A with a given value of I_A, passing the value of the output parameter O_B to service S_B (as input parameter I_B) and O_C to service S_C (as input parameter I_C). If the workflow orchestration is managed in a centralized way by the client C (e.g., C may represent a centralized workflow orchestration engine), C has to send I_A to S_A, S_A will return the output parameter values O_B and O_C to C, which C will pass on to S_B and S_C (see Figure 1 (a)). That is, the output parameter values O_B and O_C are not directly sent to the place where they are needed, but they are routed through C. This routing may not be optimal, if we assume that C is not interested in O_B and O_C (which may represent intermediary results of a composite service). Considering that C may be connected to a slow or expensive network (e.g., C may use a mobile device with a wireless network connection), the negative impacts of the centralized workflow orchestration become immediately apparent.

In order to optimize the data transmission between services, a trigger shall be installed as closely as possible to the service it will invoke. In the example before, C may

create the triggers T_A, T_B, and T_C for the invocation of the services S_A, S_B, and S_C. T_A shall be close to S_A, T_B to S_B, and T_C to S_C. As triggers allow to define the routing of output parameter values, T_A can be configured to directly send O_B to T_B and O_C to T_C, avoiding to pass these intermediary results through C (see Figure 1 (b)).

Even though it is advantageous to install triggers close to the actual services they are triggering, they may be set up on an arbitrary site. For instance, if the provider of S_B supports triggers, he may accept triggers directly in the same server running service S_B, which will result in efficient local communication between T_B and S_B. If it is not possible to colocate triggers and services within the same server, the provider of S_B may offer a separate server dedicated to triggers in his local network. Otherwise, dedicated servers may offer to host arbitrary triggers.

Using triggers, many different workflow orchestration schemes can be implemented. If all triggers are hosted by the client, it corresponds to the centralized workflow orchestration model described before. If all triggers are hosted by a dedicated server, it corresponds to passing the workflow to a dedicated orchestration engine which executes it in a centralized way. If each trigger is installed locally with the service that it will invoke, the workflow is executed in a fully decentralized way, delivering intermediary results only to those places where they are needed. Our framework does not dictate any of these settings. Therefore, it is possible to bootstrap the support for triggers by deploying dedicated servers to host triggers. With the time, service providers may start to directly support triggers in their environments (incremental deployment).

3. Defining Triggers

In this section we present more details regarding the installation of triggers. In our description we use the following abbreviations for identifying services respectively triggers:

SID: Service ID. Globally unique identifier of a service to be invoked. It consists of host, port, protocol, and local service identifier (e.g., service name and version number, depending on the protocol). SIDs can be computed from service descriptions, including grounding information.

PID: Parameter ID. Locally unique identifier for a service input or output parameter.

TID: Trigger ID. Globally unique trigger identifier. It consists of host, port, and a local trigger identifier (e.g., an integer number referring to a trigger).

In the following we present a simple API to deal with triggers in an abstract way:

CreateTrigger: Creates and installs a trigger.

Arguments:

- Destination of the trigger (host and port). `CreateTrigger` will ask the destination to set up the desired trigger.
- SID. The service to be invoked by the trigger.
- Service input parameters to wait for. Each parameter is identified by its PID. A parameter may be required or optional. The trigger will fire as soon as all required input parameters are available. As for a given input parameter multiple values may arrive before the trigger fires (while still some of the required input parameters are missing), the client has to define which values to preserve: `preserveLast` or `preserveFirst`. If values for optional input parameters arrive before the trigger fires, they will be passed to the service. After the trigger has fired, arriving input parameter values are discarded.
- Optional: Input data. For each input parameter, a default value may be provided. This value could be transmitted with `SendData` (see below), but including it in `CreateTrigger` may be more efficient and help to reduce network traffic.
- Output routing. For each output parameter (identified by a PID_O) generated by the service SID, the output routing defines a possibly empty list of pairs (TID_i, PID_i) to forward the output parameter value. I.e., whenever the service SID returns a value for the output parameter PID_O, the trigger will forward it to all triggers TID_i as input parameter PID_i, implementing a multicast. If there is a communication problem with a trigger TID_i, the trigger will retry to forward the data several times in order to overcome temporary network problems.
- Desired timeouts:
  1. Timeout to wait for input parameter values, starting with the installation of the trigger. If not all required input parameter values arrive before this timeout, the trigger will be discarded.
  2. Timeout to wait for service completion, starting with the service invocation.
  3. Timeout to wait for completed forwarding of output parameter values, starting when the trigger receives the response of the service invocation.
- Optional: Destination for failure notification message (host, port, protocol). In the case of a failure (i.e., service returning a failure message or expiration of one of the timeouts mentioned before), a failure notification is sent before the trigger is discarded, including information concerning the current state of the trigger. The level of detail of this notification can be configured. The message may simply indicate the reason of the failure, or it may include input resp. output parameter values the trigger has received so far. This information may help the client to recover from the failure.

Results:

- TID of the installed trigger (if the trigger was accepted).
- Granted timeouts. Each granted timeout may be the desired timeout or shorter.

RemoveTrigger: Explicitly removes a trigger. Normally, a trigger is removed automatically if either a timeout occurs or if the output routing task is completed, i.e., all output parameter values have been forwarded according to the trigger's routing information.

Arguments:

- TID. The trigger to remove.

SendData: Sends input parameter values to a trigger. Normally, triggers receive input parameter values either by initialization (see the input data of `CreateTrigger`) or through other triggers (forwarded output parameter values from other services). However, a client may want to install a trigger and provide input parameter values later on.

Arguments:

- TID. The trigger to send data to.
- Input data. For each input parameter, a value may be provided.

Status: Returns information concerning the status of a trigger. I.e., whether the trigger is still waiting for required input, which input parameters have been received so far, whether it has already triggered the service, whether it is waiting for the service output parameters, etc.

Arguments:

- TID. The trigger to ask for its status.

Results:

- Status information.

Three different protocols are involved in the communication with triggers and services:

1. **Trigger–service:** The trigger communicates with the service using remote procedure calls (e.g., SOAP RPC [14]). I.e., the trigger is transparent to the service, it behaves as any other client.

2. **Trigger–trigger:** The communication between triggers is unidirectional. A trigger forwards results to other triggers. The message sent from trigger T_A to trigger T_B contains at least one value for an input parameter T_B is waiting for. Even though the communication protocol between triggers need not necessarily comply with standards, SOAP messages are well suited for trigger–trigger communication. If the service invocation has failed, the trigger does not send any message on the normal output routing path, but it may generate a failure notification message (if specified in `CreateTrigger`). Subsequent triggers will notice the failure by a timeout.

3. **Client–trigger:** A dedicated, simple protocol supports the API primitives described before. For instance, `CreateTrigger` will try to set up a trigger on the specified destination platform.

In the following we present a few aspects of a prototype implementation in Java. Our prototype is based on Axis [1], an implementation of SOAP [14]. There are four different service styles in Axis: Three of them ('RPC', 'Document', and 'Wrapped') provide different ways of XML to Java binding. The fourth one, called 'Message', allows to receive and return arbitrary XML data in the SOAP envelope without any type mapping (no data binding). In our prototype, triggers use the 'Message' style, while we still assume that received messages use the 'RPC' style encoding. In this encoding, a RPC request message is modeled as an outer XML element, which matches the operation name and contains inner XML element tags mapping to service parameters. A trigger extracts all input parameters from incoming messages and merges them into a single new RPC request message to invoke the actual service. The RPC response message is split in order to create outgoing messages for each of the output parameters.

4. Decentralized Workflow Orchestration

In this section we show how a workflow can be executed using triggers. First, the client decomposes a given workflow into sequential parts that contain neither loops nor conditionals (dataflows). Each sequential workflow fragment is executed in the following way:

1. The client creates a (temporary) local service with SID S_{client} to handle the final results of the workflow fragment.

2. The client uses `CreateTrigger` to locally install a trigger to handle the workflow results, referring to S_{client}. TID_{client} is the resulting TID.

3. Starting with the last service in the sequential workflow fragment, a trigger is created for each service invocation. The output parameter values of the last service shall be routed to TID_{client}. The order of setting up the triggers ensures that for each service, the triggers of all subsequent services are created before.

4. Using the optional input data of `CreateTrigger` or `SendData`, the client sends the input parameter values of the workflow fragment to the triggers where these inputs are needed. The trigger of the first service in the workflow fragment will receive all required input parameters and execute the service. The results will be forwarded according to the trigger's output

routing, eventually triggering the execution of subsequent services. The client is not involved in this process. It is notified of the completed workflow fragment by the invocation of S_{client}.

As an example of the use of triggers, we consider a composite service built from the following simple services:

- `getGPS`: Returns the GPS coordinate of an address.
  Inputs: {`adr_text`}, Outputs: {`adr_gps`}
- `getRoute`: Computes a route between two GPS coordinates as a TIFF image.
  Inputs: {`from_gps`, `to_gps`}, Outputs: {`route_tiff`}
- `tiff2jpeg`: Converts a TIFF image to a JPEG image.
  Inputs: {`image_tiff`}, Outputs: {`image_jpeg`}

We assume that as input the client provides two addresses, `start` and `destination`. The composite service generates a map illustrating the route between these two addresses. As the client has a mobile device with limited memory and a slow wireless network connection, the map shall be delivered in JPEG format with high compression (`result`). The composite service may be described by the following workflow. The variables `tmp1`, `tmp2`, and `tmp3` are intermediary results the client is not interested in.

1. `getGPS`(`adr_text` ← `start`): (`adr_gps` → `tmp1`)
2. `getGPS`(`adr_text` ← `destination`): (`adr_gps` → `tmp2`)
3. `getRoute`(`from_gps` ← `tmp1`, `to_gps` ← `tmp2`):
   (`route_tiff` → `tmp3`)
4. `tiff2jpeg`(`image_tiff` ← `tmp3`): (`image_jpeg` → `result`)

Certainly, the workflow shall not be executed on the mobile device of the client, since this would require transferring the large uncompressed TIFF image (`tmp3`) to and from the resource-constrained mobile device. However, with the aid of triggers, the workflow can be executed without transferring any intermediary result to the client. The pseudo-code in Figure 2 illustrates how the client sets up the decentralized orchestration of the workflow. For the sake of easy readability, details, such as the negotation for timeouts, are left out intentionally.

With `createLocalService` the client simulates a local service that is able to receive the final result delivered by `tiff2jpeg`. From the client's point of view, the delivery of the result is asynchronous, as `tiff2jpeg` is not directly invoked by the client. In this example we assume that it is possible to install the triggers directly within the different server environments. The triggers are created in reverse order of the service invocation sequence in the workflow. The input parameter values for the invocations of `getGPS` are directly passed with the `CreateTrigger` primitive, i.e., the triggers `TID1` and `TID2` will fire immediately after installation. The output parameter values will be routed to the trigger `TID3`, which will wait until both `from_gps` and `to_gps` are available. Finally, `TID4` will wait for the data forwarded by `TID3` and pass the output parameter value of `tiff2jpeg` to the client (via `TID0`).

```
SID0 = createLocalService("result");
SID1 = SID2 = locateService("getGPS");
SID3 = locateService("getRoute");
SID4 = locateService("tiff2jpeg");

TID0 = CreateTrigger:
   destination = location(SID0), // dest. = localhost
   SID = SID0,
   input = [("result", required, preserveLast)],
   inputData = [],
   output = [];

TID4 = CreateTrigger:
   destination = location(SID4),
   SID = SID4,
   input = [("image_tiff", required, preserveLast)],
   inputData = [],
   output = [("image_jpeg" -> [(TID0, "result")])];

TID3 = CreateTrigger:
   destination = location(SID3),
   SID = SID3,
   input = [("from_gps", required, preserveLast),
            ("to_gps",   required, preserveLast)],
   inputData = [],
   output = [("route_tiff" -> [(TID4, "image_tiff")])];

TID2 = CreateTrigger:
   destination = location(SID2),
   SID = SID2,
   input = [("adr_text", required, preserveLast)],
   inputData = [("adr_text" <- destination)],
   output = [("adr_gps" -> [(TID3, "to_gps")])];

TID1 = CreateTrigger:
   destination = location(SID1),
   SID = SID1,
   input = [("adr_text", required, preserveLast)],
   inputData = [("adr_text" <- start)],
   output = [("adr_gps" -> [(TID3, "from_gps")])];
```

Figure 2. Executing a composite service using triggers.

5. Failure Handling

In our approach, composite web services are executed in a completely decentralized way. Therefore, it is not easily possible to monitor the progress of each service invocation. As the client will only receive the final results of the composite web service, in general it will notice a failure only after a timeout. In this case, the client may restart the execution of the workflow.

If the used web services are not reliable, this approach may result in bad overall performance, since intermediary results may have to be computed multiple times. Hence, the client should make use of the failure notification mechanism in order to collect partial results that had been computed before the failure has happened. Based on the failure notification mechanism, the client could exploit redundant execution plans in order to replace a failed web service. As an alternative (but inefficient) solution, if the decentralized

orchestration of a composite web service fails, the client could simply re-execute the workflow in a centralized fashion (fallback solution).

The client may also use the `Status` primitive of triggers in order to monitor the progress of the execution. Note that `Status` will fail if the trigger has already been removed (i.e., after a timeout or after completing its task). However, `Status` creates additional network traffic, therefore an excessive use of this primitive is not consistent with the principal idea of our approach to minimize the network traffic involving the client.

6. Security Issues

In this section we briefly discuss topics concerning security that arise due to the use of triggers. We distinguish between existing security infrastructure that may hamper the use of triggers and new security threats because of triggers.

As the placement of the triggers affects the communication paths between services and clients, firewalls may prevent the installation of triggers on certain hosts. For instance, if the client C is allowed to communicate with the services S_A and S_B, S_A may not necessarily be able to directly communicate with S_B. Therefore, a trigger installed close to S_A may fail to directly forward intermediary results from S_A to a trigger colocated with S_B.

A related problem concerns authentication. For example, S_B may want to verify that the origin of a service request is the client C. However, as triggers act as proxies that may collect input parameter values from various sources, authentication may fail. This problem could be mitigated by authenticating only the installation of the trigger, even though this does not ensure the same level of security as authenticating that all input data comes from C. Nonetheless, for the composition of information services that are open to the public, the problems concerning firewalls and authentication usually do not crop up.

The trigger infrastructure may be the target of attacks. For instance, if TIDs are not well protected, an attacker may remove a trigger or send fake data, causing the trigger to fire. Because of the forged input data, the triggered service will compute incorrect results. The client may not notice this kind of attack, because once the triggers have been set up, the client does not control the interaction between services and triggers. This problem may be addressed by using TIDs as capabilities, e.g., by chosing a large random number as a part of the TID. Then, if triggers are communicated only between trusted parties across protected (i.e., encrypted) links, forging TIDs will be very difficult.

The placement of triggers is another important issue. In general, triggers should be installed only on trusted sites, i.e., either on the client side, on the site of the service to be invoked, or on the site of a trusted third party. Otherwise, a trigger deployed on an untrusted site may disclose the collected input data and the output data generated by the triggered service, or forge input resp. output data.

Another issue are denial-of-service attacks. An attacker may create a large number of triggers with maximum timeout, he may send large amounts of input data to these triggers while still one required input parameter is missing. Thus, the triggers will have to process and store a significant amount of input data. However, in principle this problem is not much different from traditional denial-of-service attacks against web services. Services may be invoked very frequently and provided with large amounts of data. Such attacks may be mitigated by limiting the number of concurrent connections and limiting the size of message buffers. Similar techniques may be applied to triggers (i.e., limiting the number of concurrent triggers and limiting the buffer size of each trigger). Triggers may even improve load-balancing, as they are installed before the actual web service invocation happens. I.e., triggers allow the server to plan ahead the expected load in the near future.

7. Evaluation

In order to evaluate the benefits of our decentralized orchestration scheme, we simulated the network traffic (i.e., the sum of the sizes of all messages sent over the network during the execution of a workflow) caused by centralized orchestration and by decentralized orchestration using triggers.

The evaluation is based on our testbed for service composition [6], which allows to generate random, acyclic workflows, representing composite web services. Each workflow has a random number of nodes N ($3 \leq N \leq 15$). Two of them are special nodes, *START* and *END*, which represent the source of the initial input messages and the destination of the final output messages. All other nodes represent service invocations. Concerning network traffic, we consider the worst case: Each service is located on a different host, i.e., each message in the workflow generates network traffic. As *START* and *END* represent the client executing the workflow, they are located on the same host. Directed edges between nodes represent the flow of messages between the services. Each node, except for *START*, receives 1–3 input messages (*START* does not receive any input message). Each node, except for *END*, generates 1–3 output messages (*END* does not generate any output message). The concrete number of messages received and generated by a service, as well as the data dependencies between service invocations (i.e., the edges in the workflow) are chosen randomly. There is no edge between the *START* and the *END* node.

In our framework, the result of a service invocation is represented by its output parameters. When delivered to the

next services in the workflow, these parameters can be encapsulated in messages customized for each receiver or can be delivered as the same message independently of the destination. In our evaluation, we considered both possibilities: In the setting *SameMsg*, each service invocation generates a single output message which is sent to the 1–3 target nodes, whereas in the setting *DistinctMsg*, each service generates 1–3 distinct output messages.

In the case of centralized orchestration, each message transfer involves the centralized coordinator, which we assume to be located on the same host as the nodes *START* and *END*. In the setting *DistinctMsg*, each edge between two nodes that are different from *START* and *END* corresponds to two messages on the network, because the message has to be sent first to the coordinator. The edges from the *START* node as well as the edges to the *END* node correspond to a single message. In the setting *SameMsg*, the situation is different, because each service has to send only a single message to the coordinator. Moreover, a node sending a message to the *END* node may send the same message also to other nodes without requiring an extra message to the coordinator (as the *END* node is located on the same host as the coordinator).

In the case of decentralized orchestration using triggers, each edge corresponds to a single message (the edges from the *START* node correspond to client–trigger communication, all other edges represent trigger–trigger communication). We assume that each trigger is located on the same host as the service it invokes. I.e., trigger-service communication does not generate network traffic. However, client-trigger communication (in order to create triggers) causes additional messages, proportional to the number of service invocations in the workflow. For all measurements, we assume trigger creation messages to be 2KB. We do not use the input-data argument of the `CreateTrigger` primitive, but we assume that input data originating from the *START* node is sent by the `SendData` primitive. Hence, 2KB is a reasonable size for trigger creation messages.

We generated 10 000 random workflows, and for each of them, we computed the network traffic for different settings (*SameMsg* versus *DistinctMsg*), different orchestration schemes (centralized versus decentralized), and varying size of input/output messages (message size between 1KB and 150KB). Each measurement represents the average network traffic (arithmetic mean) computed over the 10 000 random workflows.

Figure 3 shows the percentage of network traffic caused by decentralized orchestration using triggers relative to the network traffic caused by centralized orchestration. The results confirm that our decentralized orchestration scheme is able to significantly reduce network traffic when compared with centralized orchestration. In the setting *SameMsg* (resp. in the setting *DistinctMsg*), decentralized orchestration causes about 76% (resp. 59%) of the network traffic due to centralized orchestration for input/output messages larger than 20KB. For smaller input/output messages, there is less reduction of network traffic in the decentralized orchestration scheme. Only for very small input/output messages (1KB), decentralized orchestration using triggers may cause extra network traffic of up to 45% (in the setting *SameMsg*), which is due to the overhead of trigger creation messages.

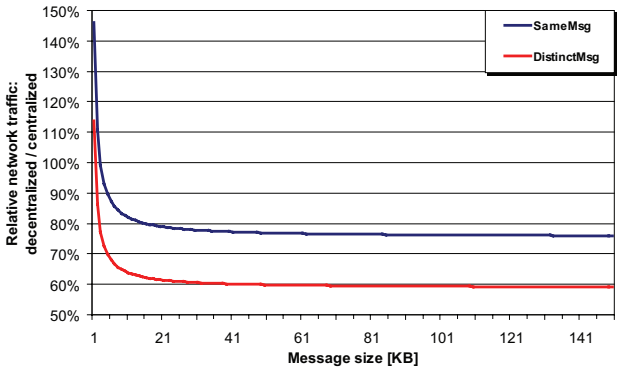

Figure 3. Percentage of network traffic with decentralized orchestration using triggers relative to centralized orchestration (average computed over 10 000 random, acyclic workflows).

8. Related Work

There is a large amount of related work concerning workflow systems and the decentralized execution of workflows. For instance, reference [11] describes a workflow trading system using mobile agents. More recently, the AMOR system [8] uses mobile agents, too. With the aid of mobile agents, it is possible to move the control of the workflow execution to different sites, which can help to reduce the network bandwidth used by communicated (intermediary) results. Moreover, mobile code enables the dynamic deployment of local data processing functions close to the data where it is needed. For example, if the client has to transform intermediary results before passing them to another service, the transformation functionality may be provided by the mobile agent which will perform the transformation where the data originates. The drawbacks of using mobile agents are increased security risks and usually a high overhead. Accepting mobile agents in the execution environment opens the doors to potentially malicious or erroneous code. Thus, our triggers currently do not support mobile code.

The internet indirection infrastructure i3 uses triggers to decouple sender and receiver [12]. In contrast to our approach, i3 triggers work on the level of individual packets

and do not support waiting conditions (synchronization) to aggregate multiple inputs from various locations before forwarding the data. i3 supports only a very limited form of service composition, where individual packets can be directed through a sequence of services. While our triggers are rather transient (used only for a single service invocation) and their placement is explicitly controlled by the client, i3 triggers are more persistent (they act as a longer-term contact point for a service) and are mapped to the Chord [13] peer-to-peer infrastructure, which allows only a limited form of optimizing the routing (by selecting a trigger identifier that will map close to a desired location). Summing up, even though there are some ideas in common, i3 has different goals (indirection, supporting mobility, multicast, anycast) and works at a much lower level than our approach. Our focus is on the efficient routing of intermediary results during the execution of composite web services.

In reference [9] the authors point out the inefficiencies of the centralized orchestration of BPEL4WS programs [3] by engines such as BPWS4J [4]. They describe an algorithm to decompose BPEL4WS programs for decentralized orchestration. In contrast to this work, which is restricted to the execution of BPEL4WS programs, our service invocation triggers provide a much more generic and lightweight infrastructure that may serve as the basis for different workflow orchestration models and engines.

The SELF-SERV system [2] focuses on web service composition. It supports peer-to-peer orchestration of composite web services without relying on a centralized coordinator. While reference [2] describes a rather complex middleware, our triggers are a lightweight solution that can be easily integrated into existing infrastructure.

9. Conclusion

If a composite web service is executed in a centralized way, intermediary results are forwarded through the site that coordinates the execution. E.g., if a client executes a composite web service on a mobile device with limited network connectivity, the transmission of the intermediary results may significantly slow down the overall execution of the composite web service, it may be expensive (costs for the caused network traffic), or it may be simply impossible if the intermediary results are too large.

In order to overcome these problems, we developed a novel infrastructure with service invocation triggers that are able to route intermediary results from their origin directly to the sites where they are consumed. Triggers act as proxies for individual web service invocations. They aggregate the input data, trigger the service execution when all required input parameters are available (synchronization), and route the service results, supporting multicasting. Based on triggers, composite web services can be executed in a completely decentralized way. Evaluation results confirm that our decentralized orchestration scheme allows to significantly reduce network traffic in comparison with centralized orchestration.

References

[1] Apache Software Foundation. Axis, http://ws.apache.org/axis/.

[2] B. Benatallah, Q. Z. Sheng, and M. Dumas. The self-serv environment for web services composition. *IEEE Internet Computing*, 7(1):40–48, 2003.

[3] BPEL4WS. Business process execution language for web services version 1.1, http://www.ibm.com/developerworks/library/ws-bpel/.

[4] BPWS4J. A platform for creating and executing BPEL4WS processes, http://www.alphaworks.ibm.com/tech/bpws4j/.

[5] I. Constantinescu, W. Binder, and B. Faltings. Flexible and efficient matchmaking and ranking in service directories. In *2005 IEEE International Conference on Web Services (ICWS-2005)*, pages 5–12, Florida, July 2005.

[6] I. Constantinescu, B. Faltings, and W. Binder. Large scale testbed for type compatible service composition. In *ICAPS 04 workshop on planning and scheduling for web and grid services*, 2004.

[7] I. Constantinescu, B. Faltings, and W. Binder. Large scale, type-compatible service composition. In *IEEE International Conference on Web Services (ICWS-2004)*, pages 506–513, San Diego, CA, USA, July 2004.

[8] K. Haller, H. Schuldt, and H.-J. Schek. Transactional Peer-to-Peer Information Processing: The AMOR Approach. In *Proceedings of the 4^{th} International Conference on Mobile Data Management (MDM'2003)*, pages 356–361, Brisbane, Australia, Jan. 2003. Springer LNCS, Vol. 2547.

[9] M. G. Nanda, S. Chandra, and V. Sarkar. Decentralizing execution of composite web services. In *OOPSLA '04: Proceedings of the 19th annual ACM SIGPLAN Conference on Object-oriented programming, systems, languages, and applications*, pages 170–187, New York, NY, USA, 2004. ACM Press.

[10] M. P. Papazoglou and D. Georgakopoulos. Introduction: Service-oriented computing. *Communications of the ACM*, 46(10):24–28, Oct. 2003.

[11] M. Schönhoff and H. Stormer. Trading workflows electronically: the ANAISOFT architecture. In *Proceedings of Datenbanksysteme in Büro, Technik und Wissenschaft (BTW'2001)*, Oldenburg, Germany, Mar. 2001.

[12] I. Stoica, D. Adkins, S. Zhuang, S. Shenker, and S. Surana. Internet indirection infrastructure. *IEEE/ACM Transactions on Networking*, 12(2):205–218, Apr. 2004.

[13] I. Stoica, R. Morris, D. Karger, F. Kaashoek, and H. Balakrishnan. Chord: A scalable Peer-To-Peer lookup service for internet applications. In R. Guerin, editor, *Proceedings of the ACM SIGCOMM 2001 Conference (SIGCOMM-01)*, volume 31, 4 of *Computer Communication Review*, pages 149–160, New York, Aug. 27–31 2001. ACM Press.

[14] W3C. Simple object access protocol (SOAP), http://www.w3.org/tr/soap/.