Supporting complex work in crowdsourcing platforms: a view from service-oriented computing

Author:
Xiao, Lu

Publication Date:
2017

DOI:

License:

Supporting Complex Work in Crowdsourcing Platforms: A View from Service-Oriented Computing

Lu Xiao

A thesis in fulfillment of the requirements for the degree of

Masters of Engineering

School of Computer Science and Engineering

Faculty of Engineering

The University of New South Wales

May 2017

THE UNIVERSITY OF NEW SOUTH WALES
Thesis/Dissertation Sheet

Surname or Family name: **Xiao**

First name: **Lu**　　　Other name/s:

Abreviation for degree as given in the University calendar: **Master**

School: **School of Computer Science and Engineering**　　　Faculty: **Faculty of Engineering**

Title: Supporting Complex Work in Crowdsourcing Platforms: A View from Service-Oriented Computing

Abstract 350 words maximum

Today, crowdsourcing is changing the way people work and solve problems - from "in-house working" to "public outsourcing". Many online crowdsourcing platforms allow the requester to advertise their tasks and help crowd workers find their jobs. However, those platforms mainly focus on the micro-task market and do not support the complex work consisting of interdependent, professional tasks. To crowdsource this work, we need a way to model professional workers, define the complex task, and a coordination mechanism to manage them.

To this end, we propose a service-oriented crowdsourcing framework in this thesis wherein (1). Each professional crowd worker is modelled as a service that can be self-described, dynamically discovered and assembled into the complex crowd work; (2). A complex crowd task is defined as a schema consisting of a set of units of work and their inter-dependencies, which can be used to get multiple crowd workers involved and guide their work; (3). The whole crowdsourcing lifecycle is divided into two phases: (i). Plan Phase - where the working plan on the advertised complex task is crowdsourced to generate the schema as mentioned earlier detailing the original advertising; and subsequently, this schema is transformed into a web service orchestration specification for the later auto-coordination; (ii). Execution Phase - where the execution of the planning result is crowdsourced to complete the original complex work as advertised through coordinating and interacting with multiple crowd workers, based on coordination and interaction protocol.

Originality Statement

'I hereby declare that this submission is my own work and to the best of my knowledge it contains no materials previously published or written by another person, or substantial proportions of material which have been accepted for the award of any other degree or diploma at UNSW or any other educational institution, except where due acknowledgement is made in the thesis. Any contribution made to the research by others, with whom I have worked at UNSW or elsewhere, is explicitly acknowledged in the thesis. I also declare that the intellectual content of this thesis is the product of my own work, except to the extent that assistance from others in the project's design and conception or in style, presentation and linguistic expression is acknowledged.'

Lu Xiao
June 12, 2017

Copyright Statement

Lu Xiao
June 12, 2017

Authenticity Statement

Lu Xiao
June 12, 2017

Abstract

Today, crowdsourcing is changing the way people work and solve problems - from "in-house working" to "public outsourcing". Many online crowdsourcing platforms allow the requester to advertise their tasks and help crowd workers find their jobs. However, those platforms mainly focus on the micro-task market and do not support the complex work consisting of interdependent, professional tasks. To crowdsource this work, we need a way to model professional workers, define the complex task, and a coordination mechanism to manage them.

To this end, we propose a service-oriented crowdsourcing framework in this thesis wherein (1). Each professional crowd worker is modelled as a service that can be self-described, dynamically discovered and assembled into the complex crowd work; (2). A complex crowd task is defined as a schema consisting of a set of units of work and their inter-dependencies, which can be used to get multiple crowd workers involved and guide their work; (3). The whole crowdsourcing lifecycle is divided into two phases: (i). Plan Phase - where the working plan on the advertised complex task is crowdsourced to generate the schema as mentioned earlier detailing the original advertising; and subsequently, this schema is transformed into a web service orchestration specification for the later auto-coordination; (ii). Execution Phase - where the execution of the planning result is crowdsourced to complete the original complex work as advertised through coordinating and interacting with multiple crowd workers, based on coordination and interaction protocol.

Acknowledgements

When I firstly started as a Master of Engineering student at the School of Computer Science and Engineering in the University of New South Wales (UNSW), I was hoping to get the real taste of research. Looking backwards, I have found myself achieved more than what I came for. Many people helped me along the path, people who I have built a strong relationship over the years both professionally and personally, and I would like to thank them.

First and foremost, I would like to thank my supervisor, Dr. Hye-Young Paik for being a great mentor and supporter during all stages of this research. I appreciate all her hard work and commitment to provide me with everything I needed to complete this study. Without her support, this work would not have been possible.

I would like to thank Dr. Jing Xu, Dr. Rosita Rastan, Mr. Nima Moghadam, and Mr. Alireza Mirsadeghi for their friendly supports. We had great times together sharing the similar painful and enjoyable experience as research students, which made my life at UNSW much easier.

I also would like to thank my family for always being on my side. Your sacrifices and encouragements have made my achievement today. Thank you for being there for me through both the good and bad times. I would not have made this far without your support and love.

Last but not least, I would like to thank myself who has not given up when encountering all kinds of difficulties in both life and work. With this milestone, I will be brave enough to face future challenges and achieve more along my career path.

Publications

Selected content of the thesis was published in:

Lu Xiao, Hye young Paik, Supporting Complex Work in Crowdsourcing Platforms: A View from Service-Oriented Computing, *23rd Australian Software Engineering Conference*, pp. 11-14, 2014.

Contents

List of Figures

Chapter 1

Introduction

Crowdsourcing has been conceptualized as a distributed problem-solving and production model for both individuals and organisations [Bra08]. It enables users to outsource their tasks or problems to the public crowd on a global scale. By utilising the collective intelligence on demand, it can solve those problems that need more of human computation and less of machine computation.

Today, there are many online crowdsourcing platforms, such as Amazon Mechanism Turk [1], acting as brokers between *task requesters* and distributed *crowd workers*. Those platforms allow *requesters* to advertise tasks with financial compensation, and help *crowd workers* find jobs.

However, most of them target at the micro-task market in which the advertised tasks are simple, independent and small-scale pieces of work, e.g. photo identification, video or other media content tagging, etc. It is insufficient for those platforms to support the crowdsourcing of creative, complex, and structured work that needs various professionals with diverse expertise and coordination of these highly skilled

---

[1]https://www.mturk.com/mturk/

individuals [ANM⁺13]. A software development or testing process, producing an academic paper or book, can exemplify these sophisticated and professional crowd work.

On the other hand, Service-Oriented Computing (SOC) has been commonly referred to as a software development paradigm to build complex and scalable systems in a distributed environment [PTDL07]. It encapsulates the functionality of each different software as a service that, in turn, specifies flexible interactions via binding. This paradigm regards the heterogeneous applications on the web as the autonomous and self-described services that can be loosely coupled and assembled.

Therefore, we can create a complex, distributed system by composing these services on a web scale instead of building it from scratch. We believe that a complex crowdsourcing platform can be seen as a large distributed system in which each task requester can be viewed as a *service requester* and each crowd worker can be treated as an autonomous *service provider*. In this sense, we could build a crowdsourcing platform that is capable of dealing with the complex crowd work mentioned above by discovering, composing, and coordinating multiple crowd workers.

1.1   Motivation

Here we describe a motivating scenario to highlight the challenges of crowdsourcing the complex work. In this scenario, a software development team wants to outsource their testing process to the crowd, due to the lack of certain resources (e.g. professional testers). As we can see in Fig. 1.1, this crowd work is structured and consists of several interdependent, professional tasks. Different tasks require different professionals. For example, to make test cases, the worker should be capable of analysing and understanding the business value or objective behind the software

product, and transforming them into the formal description of testing scenarios; similarly, to perform either functional or non-functional testing, the worker should be capable of using some testing tools or writing some auto-testing scripts.

Therefore, this crowd work is not one simple task/worker pair; instead, it needs to have multiple professionals involved and coordinate their outputs, e.g., worker A cannot start the testing work until the test case specification from worker B.
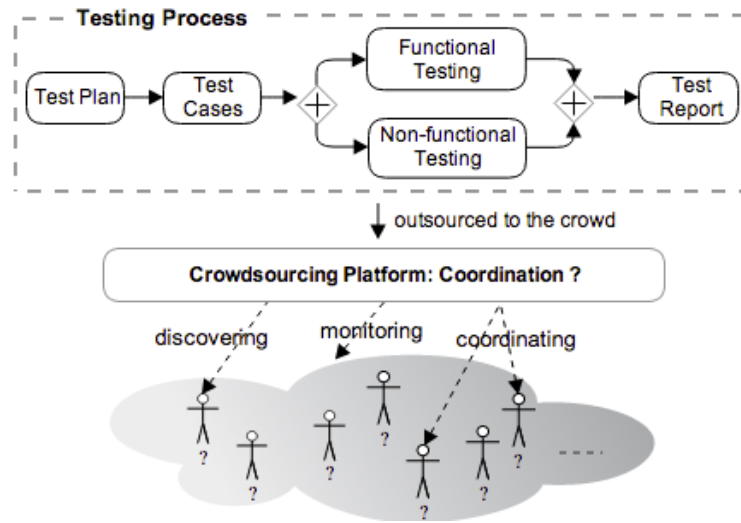
Figure 1.1: An example of crowdsourcing complex work

By crowdsourcing this process, the requester expects a platform to be capable of:

defining the complex (not simple) crowd work with expected output;

discovering the appropriate workers from the crowd, in terms of their qualification, reputation, and motivation;

coordinating crowd workers based on the dependencies between their work, e.g. the control-flow or data-flow dependency;

supervising or monitoring the performance of crowd workers, and controlling the quality of their work.

1. Introduction

We refer to the above as the *coordination requirements* for the crowdsourcing platform. To our best knowledge, most of current crowdsourcing platforms cannot meet this requirement comprehensively. Many popular ones, e.g. Mechanical Turk, Mobile-Works [2], Manpower [3], etc., mainly focus on matching or pairing a specific task with a particular crowd worker, since they target themselves as an online marketplace for job advertisers and seekers. They lack a systematic support of organisational behaviour, i.e. task decomposition, worker allocation, performance supervision, and quality assurance.

In this thesis, we present our solution to the above *coordination requirements* from a Service-Oriented Computing perspective. We propose the conceptual design of a workflow-based, service-oriented framework to better support the crowdsourcing of complex work. We show a prototype showcasing the main concepts introduced in the framework and demonstrate its use by a use case scenario.

The rest of the thesis is structured as follows:

Chapters 2 and 3 introduce some preliminary concepts and explore a wide range of related work in both industry and academia to envision the future of crowdsourcing. Chapter 4 introduces a fundamental design of concepts that underpin the framework. Chapters 5 and 6 present how the foundational concepts are realised in a particular framework named SOC4Crowd, introducing concrete data and operational models. Chapter 7 shows the implementation details of the prototype and a use case scenario to demonstrate how the system works. Finally, a conclusion and future work discussions are presented in Chapter 8.

---

[2]http://www.mobile-works.org/

[3]https://www.manpower.com.au/

Chapter 2

Preliminaries

In this chapter, we introduce some basic and broad concepts on the relevant domains of the thesis, namely: crowdsourcing and service-oriented computing. For the crowdsourcing, we present a separate literature review in Chapter 3, discussing more detailed and relevant work and systems. In this chapter, we aimed to introduce some generic concepts and common systems. For the SOC concepts, we focus on summarising the implementation technologies as they are directly applied to our implementation of a SOC-based crowdsource platform.

2.1 Crowdsourcing

Crowdsourcing is a relatively new term. It was often cited to have been firstly coined in a *Wired* magazine article by Jeff Howe[How06], which was derived from outsourcing. According to him, *Crowdsourcing is the act of taking a job traditionally performed by a designated agent (usually an employee) and outsourcing it to an undefined, generally large group of people in the form of an open call.* There are many

other attempts to define crowdsourcing and an extensive list of definitions is available in both academy and industry. As such, a variety of terminology are currently used in regard to crowdsourcing, e.g. *collective intelligence*, *human computation*, *peer production*, *mass collaboration*, *crowd wisdom*, etc.

We classify the current online crowdsourcing systems in three different dimensions - i.e. *task complexity*, *work motivation*, and *result generation*. Particularly, as summarised in Fig. 2.1,

Dimension	Value	Supported Crowdsourcing Platforms
Task Complexity	micro-task	Amazon Mechanical Turk (MTurk) ...
	complex task	CrowdFlower; Innocentive; TopCoder; 99Designs ...
Work Motivation	paid	MTurk; Innocentive; Freelancer; oDesk; ...
	unpaid	HelpFindJim; ESP Game; GalaxyZoo ...
Result Generation	integration	Google Image Labeler ...
	Selection	Threadless; Goldcorp; Innocentive ...

Figure 2.1: Crowdsourcing Classification

Task Complexity Dimension: in this dimension, we categorise the platforms according to the complexity of crowd tasks advertised;

Micro-tasks are simple and do not require much time, nor skills to perform. Typical examples can be image tagging or identification, short language translation. As a well-known platform that supports the micro-tasks, *Amazon Mechanical Turk*[1] *(MTurk)* provides an online general-purpose marketplace helping the task requester find crowd workers. MTurk coined the term *HITs (human intelligence tasks)*. As illustrated in Fig. 2.2, most of HITs are simple, independent, and only require a small amount of time from the crowd worker.

---

[1]https://www.mturk.com/mturk/welcome

6

Figure 2.2: Micro-task example in Amazon Mechanical Turk

Complex tasks often require some level of knowledge and skills, and time to perform. Examples can vary from creative tasks, e.g. product logo design, academic paper writing, to very professional, complex projects that are often scientifically or technically challenging, e.g. gold mining, a new medical treatment approach. For instance, NASA recently announced that it is turning to crowdsourcing platform for new open innovation contracts [Nor15]. As a typical platform that supports the complex, professional tasks, *Innocentive*[2] provides an online marketplace where any innovation challenge can be advertised by the *solution seeker*, while anyone can become the *problem solver* to post their solution and compete for a prize. As shown in Fig. 2.3, the problem posted on Innocentive is expertise-specific and often across multiple disciplines, e.g. *chemistry*, *engineering*, and *science* as tagged.
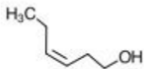
Work Motivation Dimension: in this dimension, we consider how the platforms motivate crowd workers to participate in the open requests.

Paid incentive has been proven as a direct and effective way of motivating people

---

[2]https://www.innocentive.com/

Figure 2.3: Complex task example in Innocentive

to work in the traditional employment. So it is in many commercial crowd-sourcing platforms. To financially incentivize crowd workers, there are two main patterns among most of the commercial crowdsourcing platforms.

In the first pattern, the requester sets a fixed amount of remuneration while all participating crowd workers compete against each other with their own solution for it. In the second pattern, when the requester has advertised a task, the crowd workers can give quotes to the requester. During the quoting process, each crowd worker can only see their own quote. The requester will gather all suggested quotes and choose the final worker. Platforms, such as *Freelancer*[3], *oDesk* or its rebranded *Upwork*[4], and *ServiceSeeking*[5], are typical examples of this pattern.

The payment does not have to be money. *CrowdFlower*[6], as an intermediary between businesses and workers, helps other companies utilise the crowdsourcing and rewards workers with gift certificates or virtual currency.

Unpaid motivation also exists for many people to participate in the crowd work.

---

[3]https://www.freelancer.com.au/

[4]https://www.upwork.com/

[5]https://www.serviceseeking.com.au/

[6]https://www.crowdflower.com/

Today, there are hundreds of thousands of volunteers performing unpaid tasks online, such as deciphering scanned text[7], discovering new galaxies[8], and so on. In terms of their specific motives, we see altruism, interests or entertainment as main categories.

There are many examples that show people participate in unpaid crowdwork out of altruism, one of which is the "Help Find Jim" event [Vog07]. Hellerstein et. al. reports on the experience and challenges involved in finding a computer scientist named Jim Gray who went missing during a sailing trip in early 2007 [HT11]. Thousands of online volunteers inspected thousands of satellite images in order to determine his location. Although unsuccessful in the end, the event demonstrated that people are willing to spend a significant amount of time and effort to do good when they can see the good cause.
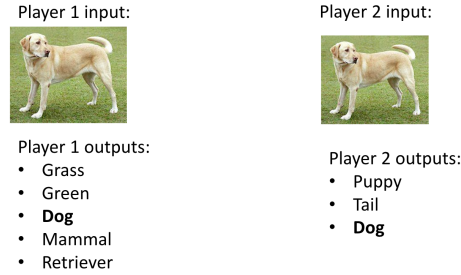
Figure 2.4: Image Recognition in ESP Game

Some crowdsourcing platforms attract volunteers through appealing to their interests and bringing in the enjoyment while they are performing the task [PRM10, FFG+14]. For example, using similar interface shown in Fig. 2.4, *ESP Game* or *Google Image Labeller* has successfully motivated many volunteers to help them improve the image recognition and search computation. Any player in the game would be arbitrarily partnered with another participant from the crowd to label the same input image. Those two bundled participants

---

[7]recaptcha.net

[8]https://www.galaxyzoo.org/

are anonymous to each other and they cannot communicate; neither of them would know the answer given by the other. Only when both of them submit the same label on the same image, then can they pass the game and continue to the next image recognition task. The players are ranked by their scores and the amount of time played.

In fact, the crowdsourcing platforms in these instances are playing a role in training complex machine learning algorithms. A typical technique for training AI systems is to feed them a very large number of labelled examples. The crowdsourcing platforms are designed to effectively generate and collect the data as training examples from the crowd workers [RH16].

Result Generation Dimension: in this dimension, we consider how the crowdsourcing result is generated in the end.

Integration: in this pattern, each crowd worker contributes to part of the final result. In other words, the working outcome of each crowd worker is integrated together into the final crowdsourcing result. A typical example can be the *Google Image Labeller* platform mentioned above, as to successfully label an image, it needs the output of both participants.

Selection: in this pattern, each crowd worker contributes their own solution as a whole to the original request, and only one or some of them will be selected by the requester as the final crowdsourcing result. As such, many crowdsourcing platforms make use of a competition strategy to get the ''best'' solution from the crowd. Examples are ranging from T-shirt design competition in *Threadless* crowdsourcing community to the research and development challenge competition in *Innocentive* crowdsourcing platform.

2.2 Service-Oriented Computing

The thesis discusses many terms and definitions originating from the Service-Oriented Computing (SOC) field. Here, we start with its definition and some important terms. Then we discuss a little further about its technical stack.

2.2.1 Definition and Terms

SOC has been commonly referred as a software development paradigm to build scalable and flexible distributed systems. In the *SOC* context, the key term *service* is a platform-independent, autonomous computational unit acting as a standalone software component. It can be self-described, published, discovered, and dynamically composed and assembled into other services. This key term reflects the idea of *Service-Oriented (SO)* approach, which aims to build a software system through discovering and invoking network-available services instead of programming it from scratch. It shifts the software development paradigm from the traditional in-house programming to cross-organisational integration [YWZ+04].

2.2.2 Structure and Key Technologies

We summarise the key technologies developed in the SOA domain as follows (shown in Fig. 2.5): there are two interaction-centric layers – i.e. *Service Foundation* and *Service Composition*, and one management-centric tier, i.e. *Service Management*, as a common aspect across the previous two layers.
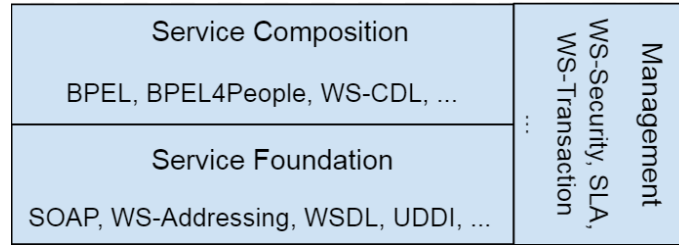
Figure 2.5: SOA Structure and Stacks of Technology

Service Foundations

The *Service Foundation* layer aims to standardize the basic interaction among the heterogeneous software systems. The two most important standards in this layer are: web service description language (WSDL) and Simple Object Access Protocol (SOAP). WSDL is an interface description language written in XML, which enables an automatic machine-processible interface declaration, discovery and bindings. SOAP provides a standard message format for sending and receiving service requests and responses. The exact content of SOAP messages are prescribed in their associated WSDL. Typically, the SOAP messages are transported via HTTP with an XML serialization in conjunction with other web service related standards such as WS-Security.

WSDL/SOAP example   To illustrate how a web service operation is described in WSDL and what SOAP interaction looks like, we use the following concat() – string concatenation – operation as an example.

The operation concat() receives two string parameters s1, s2 and returns a string which is a concatenation of s1 and s2. WSDL description of this functionality is shown in Figure 2.6 (Left). It shows that the operation named 'concat' takes one input message named 'concat' and one output message named 'contactResponse'. In turn, each message content is defined in the 'message' elements. From this, the figure

```
<?xml version='1.0' encoding='UTF-8'?>
<wsdl:definitions xmlns:xsd="http://www.w3.org/2001/XMLSchema"
  xmlns:wsdl="http://schemas.xmlsoap.org/wsdl/" xmlns:ns1="http://soa..>
  <wsdl:message name="concatResponse">
    <wsdl:part name="return" type="xsd:string">
    </wsdl:part>
  </wsdl:message>
  <wsdl:message name="concat">
    <wsdl:part name="s1" type="xsd:string">
    </wsdl:part>
    <wsdl:part name="s2" type="xsd:string">
    </wsdl:part>
  </wsdl:message>
  <wsdl:portType name="SimpleConcat">
    <wsdl:operation name="concat">
      <wsdl:input message="ns1:concat" name="concat">
    </wsdl:input>
      <wsdl:output message="ns1:concatResponse" name="concatResponse">
    </wsdl:output>
    </wsdl:operation>
  </wsdl:portType>
</wsdl:definitions>
```

```
POST http://localhost:8080/SimpleConcatRPC/concatrpc HTTP/1.1
Accept-Encoding: gzip,deflate
Content-Type: text/xml;charset=UTF-8
SOAPAction: "http://soacourse.unsw.edu.au/SimpleConcat/concat"
Content-Length: 288
Host: localhost:8080
Connection: Keep-Alive

<soapenv:Envelope
    xmlns:soapenv="http://schemas.xmlsoap.org/soap/envelope/"
    xmlns:sim="http://soacourse.unsw.edu.au/SimpleConcat/">
  <soapenv:Header/>
  <soapenv:Body>
    <sim:concat>
      <s1>Hello </s1>
      <s2>World!</s2>
    </sim:concat>
  </soapenv:Body>
</soapenv:Envelope>                              SOAP Request
```

```
HTTP/1.1 200 OK
Server: Apache-Coyote/1.1
Content-Type: text/xml;charset=UTF-8
Content-Length: 224
Date: Fri, 17 Mar 2017 04:07:05 GMT

<soapenv:Envelope
    xmlns:soapenv="http://schemas.xmlsoap.org/soap/envelope/"
    xmlns:ns1="http://soacourse.unsw.edu.au/SimpleConcat/">
  <soapenv:Header/>
  <soapenv:Body>
    <ns1:concatResponse>
      <return>Hello World!</return>
    </ns1:concatResponse>
  </soapenv:Body>                               SOAP Response
</soapenv:Envelope>
```

Figure 2.6: concat() operation using WSDL/SOAP

also shows that a SOAP request message to this operation is generated (Figure 2.6 Right, Top) and carried to the destination using a HTTP POST request. The response from the operation comes in a HTTP response containing the corresponding SOAP response.

Service Composition

Underpinned by the WSDL/SOAP standards, the Service Composition layer provides a technological solution for advanced web service interactions that involve multiple services. There are two main patterns of service composition in SOA: orchestration and choreography. We briefly explain each concept.
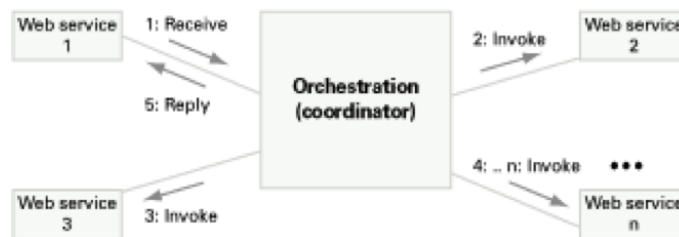
Figure 2.7: Service Orchestration Pattern

Orchestration is often regarded as a means to model and realise a complex business process within the enterprise context through composing multiple services. As shown in Fig. 2.7, there is a central orchestrator (i.e., controller) who controls the way of coordinating the interactions amongst internal and external services. In this pattern, service composition is always seen from the central orchestrator perspective, where each participating service provider is aware of the basic, end-to-end interaction between the orchestrator and itself, and not aware of other participants.

Choreography is mainly a global perspective of seeing service composition through multi-party collaboration. As shown in Fig. 2.8, there is no central controller; instead, each participant needs to properly interact with its direct partners in order to realise the global composition. To ensure correct interactions, each participant needs to conform to a common contract that specifies the message exchanges, rules of interaction and agreements that occur between parties.
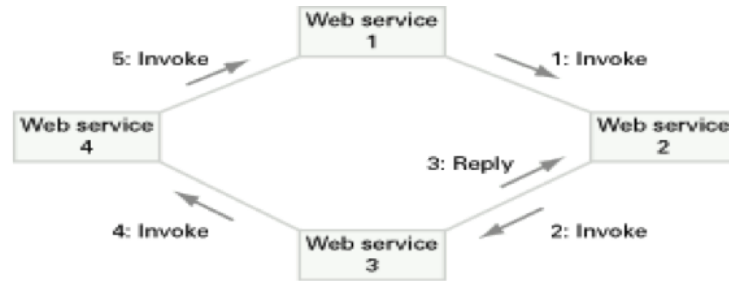
Figure 2.8: Service Choreography Pattern

Business Process Execution Language (BPEL): BPEL is one of the standard languages that implements the Service Orchestration Pattern. In the context of the thesis, the orchestration pattern provides suitable abstraction and technical implementation solutions. Hence, we introduce the language BPEL as a service orchestration language.

2. Preliminaries

The language elements in BPEL are defined as XML elements. BPEL models and realises the programming logic of orchestrating multiple services by including the following aspects of the process:

Different roles (i.e., services) that take part in message exchanges,

WSDL documents of the services and the BPEL process itself,

Control flows logic describing the coordination requirements,

Any message correlation information that defines how messages can be routed to the correct instance of the BPEL process.

The control flow definition in a BPEL process is created by combining the basic process steps called BPEL basic activities with the process structure activities called BPEL structured activities. The **Basic activities** are simple tasks and represent a unit of work/task. The **Structured activities** define the control flow. The activities represent the well-known workflow constructs such as AND-split, OR-split, AND-join, etc. Using the structured activities the explicit control flow of a BPEL process (the order in which the basic activities are executed) can be expressed. BPEL allows recursively combining the structured activities to express arbitrarily complex processes.

Let us walk through a basic BPEL example. We do not present the complete details, but include some core elements that make up an implementation of BPEL. Figure 2.9 explains the overall scenario of the example, Loan Approval Process Service. The Loan Approval Process Service is a web service that provides a port type: `LoanApprovalPT`, which has an operation named `approve`(). The client is expected to send an input message called `creditInformationMessage` and the service to return `approvalMessage`.

The internal implementation of Loan Approval Process Service is defined as a BPEL process that partners with another web service that happens to provide the same
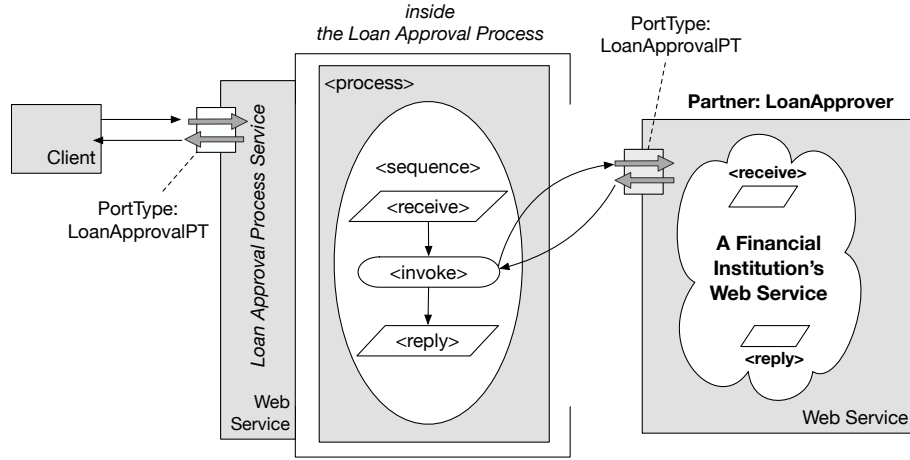
Figure 2.9: Loan Approval Process as BPEL

port type and operation: `LoanApprovalPT.approve`(). The control flow logic of the
BPEL process is as follows: the process, first, receives a `creditInformationMessage`
from the client; then, invokes the partner's approve() operation (here, passing on
the same message `creditInformationMessage`); note that the synchronous invoke
activity will receive the output message `approvalMessage` from the partner ser-
vice; finally, replies to the client passing on the `approvalMessage`. To implement
this, first the necessary messages are defined in the WSDLs of BPEL participants.
Now, Loan Approval Process Service, which is a BPEL process, is going to define a
WSDL file to expose itself as a web service. This is shown in Figure 2.10, where: (i)
the import statement brings the definition of the `LoanApprovalPT` port type into
this WSDL, (ii) a partner link type with a role `approver` is defined; the partner
that plays this role will support the `LoanApprovalPT` port type (i.e., will perform
`approve`()) ; (iii) then, the binding and service definitions for the Loan Approval
Process service are defined. These definitions are directly referenced in the BPEL
code. Let us briefly examine the BPEL code in two parts. First, in Figure 2.11,
there is a partner link definition. There are two partners to represent according to
the scenario depicted in Figure 2.9. One partner link is to represent the interactions
between the client and the Loan Approval Process (the BPEL process). The other

```
 2   <definitions name="LoanApprovalService"
 3     ...
 4     targetNamespace="http://soacourse.unsw.edu.au/loanapproval"
 5     xmlns:tns="http://soacourse.unsw.edu.au/loanapproval"
 6     xmlns:approver="http://soacourse.unsw.edu.au/loanapprover">
 7
 8     <import namespace="http://soacourse.unsw.edu.au/loanapprover"
 9       location="loanapprover.wsdl" />
10
11     <plnk:partnerLinkType name="loanApprovalLinkType">
12       <plnk:role name="approver" portType="approver:loanApprovalPT" />
13     </plnk:partnerLinkType>
14
15     <binding name="LoanApprovalServiceBinding" type="approver:loanApprovalPT">
16       <soap:binding style="document"
17         transport="http://schemas.xmlsoap.org/soap/http" />
18       <operation name="approve">
19         <soap:operation soapAction="http://soacourse.unsw.edu.au/loanapproval/approve" />
20         <input>
21           <soap:body use="literal" />
22         </input>
23         <output>
24           <soap:body use="literal" />
25         </output>
26       </operation>
27     </binding>
28
29     <service name="LoanApprovalServiceProcess">
30       <port name="LoanApprovalProcessPort" binding="tns:LoanApprovalServiceBinding">
31         <soap:address
32           location="http://localhost:6060/ode/processes/LoanApprovalServiceProcess" />
33       </port>
34     </service>
35   </definitions>
```

Figure 2.10: Loan Approval Process: The BPEL process' WSDL

link is to represent interactions between the Loan Approval Process (the BPEL process) and the partner service. These are declared as two partner links, `client` and `approver` respectively. In the client link, `myRole` = "`approver`" specifies that it is the Loan Approval Process that plays the approver role (providing the referenced port type), whereas in the approver link, it is the partner service that plays the role. Besides the partner link definition, there are two BPEL variables declared to hold the input/output messages. The actual control flow part of the BPEL process is shown in Figure 2.12. The orchestration logic starts with a `receive` activity (through the client link), then proceeds with an `invoke` activity (through the approver link) to call the partner service, and then with a `reply` activity (through the client link) to relay the response message. Thus a single pipeline of three activities is constructed with the BPEL-structured activity `sequence`.

In this chapter, we have introduced some basic concepts and technologies in both

```
1   <!-- SimpleHomeLoan BPEL Process [Generated by the Eclipse BPEL Designer] -->
2   <bpel:process name="SimpleHomeLoan"
3           targetNamespace="http://soacourse.unsw.edu.au/loanapproval"
4           suppressJoinFailure="yes"
5           xmlns:tns="http://soacourse.unsw.edu.au/loanapproval"
6           xmlns:ns1="http://soacourse.unsw.edu.au/loanapprover"
7           xmlns:ns2="http://soacourse.unsw.edu.au/loandefinitions"
8           xmlns:bpel="http://docs.oasis-open.org/wsbpel/2.0/process/executable">
9
10    <bpel:import location="loanapprovalArtifacts.wsdl" namespace="http://soacourse.unsw.edu.au/loanapproval"
11          importType="http://schemas.xmlsoap.org/wsdl/" />
12    <bpel:import namespace="http://soacourse.unsw.edu.au/loanapprover"
13          location="loanapprover.wsdl" importType="http://schemas.xmlsoap.org/wsdl/" />
14
15      <bpel:partnerLinks>
16          <bpel:partnerLink name="client" partnerLinkType="tns:loanApprovalLinkType" myRole="approver" />
17          <bpel:partnerLink name="approver" partnerLinkType="tns:loanApprovalLinkType" partnerRole="approver" />
18      </bpel:partnerLinks>
19
20      <bpel:variables>
21          <bpel:variable name="request"
22                  messageType="ns1:approve"/>
23
24          <bpel:variable name="approvalInfo"
25                  messageType="ns1:approveResponse"/>
26
27      </bpel:variables>
```

Figure 2.11: Loan Approval Process: The BPEL process (Part 1)

```
29          <!-- ============================================================ -->
30          <!-- ORCHESTRATION LOGIC                                          -->
31          <!-- Set of activities coordinating the flow of messages across the  -->
32          <!-- services integrated within this business process            -->
33          <!-- ============================================================ -->
34          <bpel:sequence name="main">
35
36              <bpel:receive name="receiveInput" partnerLink="client"
37                      portType="ns1:loanApprovalPT"
38                      operation="approve" createInstance="yes" variable="request"/>
39
40              <bpel:invoke name="Invoke" partnerLink="approver"
41                  portType="ns1:loanApprovalPT"
42                  operation="approve" inputVariable="request"
43                  outputVariable="approvalInfo" />
44
45              <bpel:reply name="replyOutput"
46                      partnerLink="client"
47                      portType="ns1:loanApprovalPT"
48                      operation="approve"
49                      variable="approvalInfo" />
50          </bpel:sequence>
51      </bpel:process>
```

Figure 2.12: Loan Approval Process: The BPEL process (Part 2)

crowdsourcing and service-oriented computing fields. In doing so, we provide an overview of our problem and solution domains, respectively. Next, we will explore the state of the art in crowdsourcing.

Chapter 3

Literature Review

Crowdsourcing has become an emerging field and attracted the attention from both academy and industry in recent years. In this chapter, we categorise the related research work and discuss each category in details. In doing so, we aim to present the state of the art in crowdsourcing domain and identify the place where our work resides in.

3.1 Domain-specific Crowdsourcing

The work discussed in this section shows how crowdsourcing, as a problem-solving model, has been applied to many different application-specific domains.

3.1.1 Crowd Ideation

The scale and diversity of the crowd make it naturally good at generating a great amount of ideas. As such in industry, there are many online crowdsourcing sites,

e.g. Quirky[1], openIDEO[2], etc., which have been built to collect numerous ideas or solutions to various problems. In academia, there are also some research work that aims to study and improve the process of crowd-based ideation or innovation.

Ideation Quality

According to [JSD16], the quality of the crowd ideation is not always matched with its quantity. For instance, there are some duplicated ideas submitted from the crowd and some of them are too vague to be understood or too impractical to be implemented. [JSD16] tries to improve the quality of crowd ideation by introducing a real-time expert guidance. They adapt the expert facilitation strategy from the face-to-face brainstorming into their crowd ideation system - i.e. IdeaGens, in which a skilled facilitator can monitor the incoming ideas on a dashboard and identify the promising ones; then (s)he can provide the high-level guidance, e.g. questions or provocations, to inspire those ideators further.

Ideation Request Representation

Ideation sometimes can be difficult, especially when the original problem is domain specific. Some researchers found that people can become more creative at problem solving, when they are inspired by the relevant examples from the outside of problem domain. Those domains that can provide inspiring examples are coined as *distance domains*. Yet it is difficult to find the outside-the-box inspiration, as people tend to become fixated on surface-level details when they are seeking the analogy from other distance domains. [LAK16] has presented a schematic approach to re-represent the original crowd problem. It abstracts the problem description through removing

---

[1]https://www.quirky.com

[2]http://openideo.com

the surface-level details, e.g. domain-specific vocabulary, and keeping the problem structure. In doing so, it reduces peoples fixation on the surface-level features of original problem and makes it easier to identify the distance domains containing the solutions to problems with the similar structure. Their follow-up work [LKA16] improved this crowd problem re-representation approach through eliciting those constraints inherent in the original problem from the crowd and abstracting them into the problem schema.

3.1.2 Crowd Design

Crowdsourcing also can be utilized for feedback generation.This has been studied in the crowd design topic.

Structured Crowd Feedback

A diverse set of feedback from the crowd can potentially improve the design; yet, sometimes, it is difficult for the designer to understand the perception of the crowd, given its unstructured comments. Further, the empirical studies conducted by some researchers indicate that, the unstructured format tends to generate low-quality feedback. To improve the quality, [AB14] has presented a system - i.e. *Voyant*, which aims to structure the crowd feedback through providing a set of coordinated views - i.e. *Elements*, *First Notice*, *Impression*, *Guidelines*, and *Goals*. Those views can help the designer understand the perception of crowd on the original design from different perspectives. Similarly, [KJLW+15] has also presented their web-based system - i.e. *CrowdCrit*, to structure design critiques from the crowd through its own views. Moreover, it also aggregates the diverse crowd critiques and visualizes the aggregation result for the designers review.

Crowd as a Part of Design Process

Other than utilizing the crowd for the one-off feedback generation, some research work has considered the crowd as part of the design process. [AHDB15] has conducted an empirical study to see if the crowd has any effect on the design process. According to their findings, crowd feedback does prompt the designer to make both deep, e.g. theme and layout, and cosmetic, e.g. font and color, changes in a design. As such, they believe that the crowd should be part of the iterative design process to improve its quality and produce more effective solutions.

When compared with their empirical study, [PKLSH13] has put "*crowd as part of design process*" into practice. Specifically, it has proposed a novel design process, coined as *Crowd vs. Crowd (CvC)*, wherein multiple design teams are formed from the crowd and compete with each other to produce a high-quality design. According to [PKLSH13], the nature of design process consists of collaboration and competition. As such, the crowd can naturally contribute to each of them. In their proposed *CvC* process, a design team is formed by a designer and his/her supporters in the crowd who like his/her original design. Within each team, members share the information and ideas, and collaborate with each other to produce their own design. Then those multiple teams compete with each other to select the final design result. Therefore, *CvC* allows the crowd to participate in the whole design process, e.g. ideation, specific design work, feedback, while it also involves the designer to coordinate and guide them for the output generation.

3.1.3 Crowd Funding

The crowdsouring platform has also been applied to generate funds for projects [FPO⁺11]. The term - i.e. *Crowdfunding*, is often referred to those online platforms

where the entrepreneur can raise funds from thousands of small investors instead of the banks or venture capitalists. In industry, there are many websites built to that end and some typical ones are Kickstarter[3], Indiegogo[4], Kiva[5], DonorsChoose[6], etc.

On most of those sites, the entrepreneur, which is often coined as *Proposer*, proposes his/her idea or project to raise funds from the crowd. When the amount of funds meets the target specified by the proposer, (s)he is awarded with the collected funds and carry out his/her project. Once the project is completed successfully, the crowd investors are rewarded in the following two forms.

Public value: everyone that is not limited to the investors can get the benefit, e.g. medical research;

Private value: only the investors can benefit from the project result, e.g. product pre-order.

In academia, some research work has studied how to increase the chance of crowd-funding success. Particularly, [MMJ+16] has conducted a quantitative analysis to link social ties with crowdfunding. Their work shows that multiple co-proposers in a team are more likely to achieve their crowdfunding goal than those who seldom propose their project on their own. Given the crowdfunding is a new form of raising funds, many work has been conducted to study it and provide various suggestions for its future design from diverse perspectives [BLS14].

---

[3]https://www.kickstarter.com/

[4]https://www.indiegogo.com/

[5]https://www.kiva.org/

[6]https://www.donorschoose.org/

3. Literature Review

Models for Crowd Funding

Many crowdfunding sites are designed in the well-known ''*All-or-Nothing*" model along with a given deadline to coordinate the crowdfunding project. *All-or-Nothing* style requires the project to hit the funding target before a deadline; otherwise all collected funds return to donors. However, this design style encourages the potential donors to withhold their donations, because they often tend to wait for the last minute to see if others have expressed their interests and values. As such, they send the mistaken signal to each other about the interests of crowd on the crowdfunding project, which causes the donation inefficiency and even failure. To mitigate this side effect, [JWR15] suggests crowdfunding sites designed in *All-or-Nothing* model to explore and develop more features to encourage early donations. Some of the future design suggestions are: i). setting a mandate pace for donations; ii). hiding the total amount of collected funds and rewarding the early donors with the extra funds that are beyond the original goal.

Proposer

[EJMG15] has conducted a quantitative study on whether or not the crowdfunding can influence the entrepreneurial self-efficacy (ESE). ESE describes the belief someone holds in succeeding at their entrepreneurship. According to their study, a well-designed crowdfunding platform can increase ESE through:

Public validation with both financial and emotional support from the crowd;
Role modelling with the access to examples and lessons of other entrepreneurs;
Mastery with the development of new skills;
Psychological state via the way that the entrepreneur can feel energized.

Among a variety of proposers on the crowdfunding sites, there is a special group of them - i.e. scientists. Today, crowdfunding science is a new way of supporting scientific research, given the traditional research support, e.g. grants and fellowships, is decreasing. Yet, less is known about how this new way affects scientists and their work. To this end, [HG15] has conducted a qualitative study on why and how scientists use crowdfunding to support their research. According to their analysis, scientists are motivated to use crowdfunding to experience the sense of

Competence through acting as a domain expert;

Relatedness through connecting to other scientists;

Autonomy through the access to financial resources without applying to the traditional fundings.

Also, they have found the current general-purpose crowdfunding sites cannot sufficiently support scientists. Scientists do not normally communicate their work to the general public. Unlike the general-purpose crowdfunding project whose outcome, e.g. products or services, can be tangibly shared and enjoyed by the crowd supporters, the outcome of scientific research is often furthering knowledge, which makes it difficult to gain the support from crowd. As such, the authors provide design suggestions for the science-crowdfunding platforms, e.g. a wide range of data visualization, communication strategies between scientists and the general public.

3.1.4 Mobile Crowdsourcing

Recently, the crowdsourcing platform has been applied to linking the requester to physical service providers. Typical examples can be picking up or dropping off some items at a certain location, reporting the queue time in a restaurant, running some household errands, etc. Most of those physical services are location-sensitive,

requiring crowd workers to use their mobile phones. Therefore, they have been coined as *Mobile Crowdsourcing*. The platforms such as Gigwalk[7] and Taskrabbit[8] act as an intermediary between the task requester and mobile users from the crowd.

In academia, [TASF+16] have presented a mobile crowdsourcing platform - i.e. *TA$Ker*, to empirically study on how the mobile crowd responds to both *Pull* and *Push* strategies used in the task selection or recommendation. In their work, the *Pull* strategy represents the pattern wherein the mobile crowd select the available task on their own, while the *Push* strategy denotes the model wherein the mobile crowdsourcing platform recommends a task to an individual worker, based on his/her current location or movement pattern. According to their analysis, the latter outperforms the former in terms of not only task selection, but also task completion.

Similarly, [JTB15] proposes a task selection model within mobile crowdsourcing. Particularly, they try to see if geography could have any impact on the task selection, e.g. the relationship between the location of a task and its price, as well as the willingness of mobile crowd performing it. To that end, they have conducted both quantitative and qualitative analyses to reveal the geographical factors that influence the mobile crowdsourcing market. In particular, they have found that the distance to a location and the socioeconomic status (SES) of a task area have a significant impact on the willingness of mobile crowd accepting the task and its cost.

Crowdsourcing as a problem-solving approach has been applied into a variety of domains, which are not limited to the aforementioned. For instance, [ERC14] and [SJ16] have applied crowdsourcing into the natural language process (NLP) field, e.g. the language grammar and styling error correction, text annotation, etc. [SZX+15] leverages crowdsourcing for the traditional eye tracking, and [ELMG16] has studied

---

[7]http://www.gigwalk.com/

[8]https://www.taskrabbit.com/

how the crowd can help the behaviour change plan generation, to name but a few.

3.2 Crowdsourcing Elements

The work in this section studies the foundational components of crowdsourcing namely: *crowd worker* and *crowd task*, regardless of any specific application domain.

3.2.1 Crowd Worker

There is a great amount of research work on crowd workers and most of them view the crowd from the perspectives of their motivation and productivity.

Motivation

Among those work studying crowd workers, a large portion focus on their motivation. [CED14] has found the intrinsic enjoyment and competition is a widely-used motivation strategy to maintain the engagement of crowd volunteers in the volunteer-based crowdsourcing project. They have also found, while the competition motivates the high-performing participants, it has a neutral or even demotivating effect on others. For instance, many mid-level performers would rather prefer to stay at the middle level and the altruism may be potentially reduced by the competition. As such, they have proposed an alternative approach coined as *normification*, through exploring theories of the social and personal norm. This approach aims to encourage the crowd to imitate each other and behave similarly. They suggest the crowdsourcing system designer to consider this approach along with competition together as the motivation strategy and each of them can be used accordingly, depending on the crowd and project. For instance, if the crowd consists of a set of exceptional performers

and the project aims to produce a high-quality result, the competition should be used to motivate the crowd. Yet, if the crowd is a collection of ordinary members and they are participating in a general-purpose project only requiring a collective effort, the normification should be considered instead.

Similarly, [MST$^+$15] also studies the motivation in the volunteer-based crowdsourcing. Yet, their target domain is more specific - i.e. social-purpose crowdsourcing. Typical crowd work in that domain can be the volunteer service for *GLAM* - i.e. Galleries, Libraries, Archives, and Museums. Just like any other volunteer-based crowdsourcing projects, it is quite challenging to sustain those volunteer services in the social-purpose crowdsourcing. As such, [MST$^+$15] has firstly analysed the characteristics of those participants in the social-purpose crowdsourcing. Then they have presented a four-quadrant model as metrics to assess the motivation of those workers in the social-purpose crowdsourcing. According to their findings, senior citizens play a primary role in the social-purpose crowd work and they are often more dedicated than the young workers. Further, the intrinsic motivations, e.g. the sense of social contribution, the feeling of community identification, etc., are the main drivers of those senior citizens participating in the social-purpose crowd work.

Productivity

When compared with the above research on the crowd motivation, other work has been studying how to increase the work quality and productivity of crowd workers. To that end, [Har15] has applied the *pay-to-quit* incentive approach from the traditional employment into crowdsourcing, in order to encourage those workers who have a poor performance to withdraw from crowd tasks at the early stage, while retaining the high-performing crowd workers. This *pay-to-quit* incentive approach is on the premise that those who are motivated by the personal finance reward only would not have the same commitment or dedication as those who are not. Through

several empirical experiments, [Har15] has found that this approach can separate the poorly-performing crowd workers from the high-performing ones, which in turn increase the mean task accuracy and productivity.

Apart from the above motivation and productivity perspectives, some work is viewing crowd workers from other interesting angles. For instance, [KMB⁺15] has been viewing a special group of crowd workers - i.e. people with disabilities. They explore the crowd to see if people with disabilities are currently involved and whether or not the current crowdsourcing platforms are feasible enough for them to effectively participate in the crowd work. According to their findings, people with disabilities are currently participating in the crowd work even though with challenges, e.g. accessibility. As such, they provide some design suggestions, e.g. more login access options, for the future crowdsourcing platforms that should consider people with disabilities.

3.2.2 Crowd Task

The work on the crowd task falls into categories of task representation, assignment, and execution.

Task Representation and Assignment

In terms of the task modelling and selection, [ECRSS16] has found, in most of current crowdsourcing platforms, the private information of a micro-task is at risk of leakage. For instance, a malicious crowd worker may be able to decipher the prescription of a medicine bottle while performing an image labelling task. To mitigate this risk, they have proposed an approach to split the crowd task into multiple components, each of which does not leak any private or confidential information and can be

independently completed by a crowd worker. Further, they also present a hybrid task assignment approach on top of the traditional *PULL* and *PUSH* strategies to avoid the collusion among crowd workers who try to collectively decipher the private information of a task.

Task Execution

In terms of the task execution, [PFSM16b] has found, from time to time, the requester experiences the long-tail execution of a crowd task because of the abandoned task assignment - i.e. some tasks are left unfinished. This happens due to various reasons, e.g. the crowd worker finds it difficult to proceed or lacks the confidence of task performance. Those abandoned task assignments result in the task execution delay, which in turn makes it difficult to predict and analyse the overall execution time. As such, they have presented an approach named *ReLauncher* to identify those abandoned assignments and relaunch them for other crowd workers during runtime. In doing so, it speeds up the overall task execution.

Future Task Design

Some research work tries to look ahead into the future of crowd work or task design and many of them believe that future lies in integrating the crowdsourcing into the organisation setting. [MOS⁺14] believes the organisation and business can leverage crowdsourcing in various ways to support their goals and suggests that, three themes of crowdsourcing - i.e. crowd work, workers, and system, are all worth being explored. In terms of the crowd work particularly, they believe the future research should try to answer the question - i.e. *How do existing models of crowd work account for the real nature and inherent complexity of organizational work?*. Similarly, [Obi15] also believes most of current crowdsourcing platforms target at

the micro-tasks which can hardly represent the complex and rich nature of organisational work. Yet, they try to shed a light on this challenge through outlining ways wherein researchers can examine and utilize the traditional work design theories for organisational crowd work.

3.3   Crowdsourcing Complex Work

The need for supporting complex work in crowdsource platforms has been recognised in both industry and academia. We present some initial work in this section.

In industry, websites like *Freelancer*[9] and *Upwork*[10] allow the requester to advertise a professional and complex task, e.g. a website development. These sites mainly aim to match a specific task with a particular crowd worker. Yet, given a complex task consisting of many interdependent work items that require multiple workers involved, current systems lack enough features to support such functions.

3.3.1   Requirements for Complex Work

In academia, some researchers try to get a glimpse of the future crowdsourcing platform in which the complex crowd task is well supported. The work introduced by [ANM+13] has envisioned the future crowdsourcing at a high level - that is, it would be a promising online workspace where the next generation would thrive on. In order to fulfill that promising vision, they have proposed an analytical framework consisting of several research foci through the analogy from both organisational behaviour and distributed computing domains. Some of those foci, for instance,

---

[9]https://www.freelancer.com.au

[10]https://www.upwork.com/

are crowd workflow, task assignment, quality control, job design, worker motivation and reputation, etc. Through progressing on those foci, they believe this analytical framework can guide both researchers and practitioners to contribute and build a future crowdsourcing platform in which the complex and creative task is supported, as well as the needs of both crowd workers and requester are considered.

Similarly, in [PFSM16a], some concrete requirements have been elicited to support the structured work crowdsourcing. To be more specific, they firstly have coined those complex, structured crowd work as *Crowdsourcing Process* that requires the coordination of multiple tasks and participants. Then they have elicited and analyzed a set of concrete requirements that need to be met by the future crowdsourcing system, e.g. crowdsourcing process definition language, control-flow and data-flow management, etc.

3.3.2 Current Approaches

To realise the above requirements and vision of future crowdsourcing, some initial work has been done. A general-purpose framework named as *CrowdForge* has been introduced in [ABSK11]. *CrowdForge* aims to support the complex and interdependent tasks crowdsourcing in the existing micro-task market. By drawing the analogy from the distributed computing domain, they have realised their own *MapReduce* approach into *CrowdForge* for the complex crowd task decomposition and combination. Specifically, they have defined a three-phase crowdsourcing lifecycle for the complex task as follows.

Partition: in this phase, a complex task is partitioned into a set of micro-tasks by the market and each of them is simple enough for a crowd worker to complete in a short amount of time.

Map: during this phase, each partitioned sub-task has been mapped to a crowd worker and generated its output.

Reduce: the output of each sub-task is merged into the final result of original complex task.

Moreover, they have presented a crowd task management tool named as *Crowd-Weaver* as their further work in [ASPR12]. *CrowdWeaver* allows the requester to monitor the crowdsourcing progress as it can visualize the execution of task flow. And it can notify the requester when any change occurs in terms of task latency, price, and quality.

A tool named as *Turkomatic* has been presented in [AMB12] aiming to improve the quality of complex crowd task or workflow design. The idea behind *Turkomatic* is that responsibility of crowd workflow design should be shared by both crowd workers and requester. Through enabling them to work together on the workflow design, the task would be defined more clearly and both parties would be on the same page, regarding its objective. To that end, *Turkomatic* has been realised as a collaborative crowdsourcing workflow design tool. In particular, it firstly allows the requester to define a crowd task with a broad objective. Then it allows the market to design the task workflow; in the meantime, it allows the requester to monitor and edit the workflow design as they produce. Once a particular part of the design process is altered by the requester, e.g. a sub-task, the updated part and its direct association would be re-designed automatically.

We draw an inspiration from these task decomposition approaches in the above research work. However, due to the nature of micro-task market wherein most crowd workers are unskilled, some complex tasks are not easy to be decomposed into a set of micro-tasks, as they often require multiple experts with the deep and domain-specific knowledge.

With the same concern, the work in [DSA⁺14] aims to get multiple, distributed experts from the crowd to form a team for collaboration and coordination. In general, they introduce their *flash teams* - i.e. a framework to dynamically assemble multiple crowd experts into a virtual team for a given complex task. In particular, they firstly allow the requester to define the complex crowd work as a set of modular tasks. Those modular tasks are chained with each other, given their input and output data-flow. Also, each of those modular tasks has a tag to be associated with multiple experts from the crowd. In doing so, multiple experts are involved for a given modular task to form what they have coined as *block*. Given the sequential data-flow, each *block* can be connected with another to form an elastic *flash team*, in order to tackle the original complex crowd work. To manage a *flash team*, they have introduced their *Foundry* framework to allow the requester to author modular tasks, automatically manage the data-flow handoff, notify the crowd experts, visualize the runtime progress of the complex crowd work, etc.

To provide a better coordination which is not limited to the above sequential data-flow management, the work in [SFPF15] has proposed a prototype solution to view the *Crowdsourcing Process* from BPM perspective. In general, their solution consists of a process modelling language - i.e. *BPMN4Crowd*, equipped with a visual editor, and its runtime platform. In particular, their *BPMN4Crowd* brings the crowd context as an extension into *BPMN* and it has defined some crowd-specific elements, e.g. *Pick Task*, *Create Instance*, *Receive Result*, *Validate Result*, and *Reward Result*, etc. Additionally, they have enumerated several crowdsourcing patterns or what they have coined as *Tactics* models - e.g. *Marketplace*, *Contest*, *Auction*, etc. Also, they have enumerated some validation and rewarding logics - e.g. *Expert Validating*, *Marking*, *Gold Data*, *Agreement*, etc. Therefore, with those extracted patterns and provided *BPMN4Crowd* language, they enable anyone with BPM background to model a crowdsourcing process. Further, in order to automate that model execution, they also provide a BPMN4Crowd runtime - i.e. *Crowd Computer (CC)*, to

manage both control-flow and data-flow dependencies in a crowdsourcing process.

3.3.3 Key Challenges

Through the above initial work stepping into the future crowdsourcing, we have found three key challenges in order to systematically support the complex work crowdsourcing. Those challenges are:

How to model the complex crowd task

How to model the professional crowd worker

How to coordinate multiple professional workers to complete the complex task

To our best knowledge, none of the above initial work has tackled all of those challenges in a comprehensive and systematic manner. For instance, authors of [ABSK11] and [AMB12] acknowledge that their crowd task decomposition approaches are limited due to the micro-task market wherein most of crowd workers are unskilled. Therefore, due to the lack of professional crowd workers, some complex tasks requiring a certain amount of domain-specific knowledge cannot be decomposed into a set of micro-tasks through their approaches. To involve professional workers from the crowd, the work in [DSA+14] has utilized the pool of professional workers in a professional crowd site - i.e. oDesk (re-branded as Upwork[11]), for the complex task. Yet, they have simplified the coordination of multiple professionals - that is, their coordination approach mainly focus on the management of sequential data-flow dependencies among professional workers and it is limited to other dependencies, e.g. parallelism and branching control-flow dependencies. To better define the work dependencies, a crowd task modelling language - i.e. *BPMN4Crowd*, has been proposed in [SFPF15]. To use *BPMN4Crowd* language, however, the requester

---

[11]https://www.upwork.com/

is required to have *BPM* background and knowledge, which raise the threshold to non-BPM users. Further, they also do not have any approach to model professional crowd workers. Some research work in the Business Process Management area tried to address the problem from the view of the users who are not familiar with BPM technologies. For example, [BBA12] uses mashup techniques, a user-focused Web technology. [WPB13] employs a visual composition language approach and represent the control/data flows as documents and linking documents.

To model human workers, a *Human-Provided Services (HPSs)* concept has been proposed in [DSBB10] and [DFS12] to model human activities in the web service format - i.e. *WSDL*. In doing so, they can leverage the mature SOA technologies, e.g. service discovery and interaction, for the expert finding and interaction. Similarly, a *Work as a Service (WaaS)* concept has been proposed in [RYMOV12] to model the large and complex human work in a distributed organisation setting. In particular, they define any piece of work as two parts of information - i.e. *payload* and *coordination*, which provides both data-flow and control-flow for management. Even though those recent research work does not target at the crowdsourcing domain, we are still inspired by their lens of looking at professional workers and complex human work from *SOC* perspective.

We believe there is an analogy that can be drawn from *SOC* field to address those three key challenges as above, in order to better support the complex work crowdsourcing. In general, if we could represent each distributed professional worker from the crowd as a service and utilize this professional crowd to define the complex task as a set of structural work items that can be linked with those human services, we then are able to apply the mature *SOC* technologies to coordinate and manage multiple professional workers to complete a complex crowd task.

Chapter 4

Conceptual Framework for Crowdsourcing Complex Work

In this chapter, we set out to identify and describe main conceptual ideas that underpin a crowdsourcing framework for complex work.

To provide a solution to the *coordination requirements* highlighted in Chapter 1, we see the following three key elements as the main components in the new crowdsourcing domain for complex work. These elements are extensions of the currently available concepts in crowdsourcing platforms, but re-designed particularly from the service orientation viewpoints.

We briefly introduce the following key concepts in our approach before detailing them in the rest of this chapter.

- *Crowd Workers*: We consider crowd workers as the providers of services in our framework. Currently, the crowd workers in many of existing platforms are represented by a simple set of descriptions with a specific focus on representing

historical performance ratings. In our framework, we take a broader view in crowd worker representations. We propose a concept called **Crowd Workers as Services (CWS)** in which we represent each individual crowd worker as a full fledged service provider with a complete service description profile. The profile, in turn, is a composition of various data services that provide a more comprehensive view about the worker. For example, one of such data services would provide the capabilities of the worker to see what kind of services the worker can perform. More importantly, by abstracting each provider in the similar interface, we could draw an analogy from SOC to meet the coordination requirement above through service discovery and composition.

- *Complex Crowd Work*: Additionally to a micro task, our framework is able to express the make up of a complex piece of work. In SOC, single service operations could be considered 'a single unit of work', providing a building block for achieving more complex application logic. We propose **Complex Crowd Task as a Schema (CCTS)** in which we define the complex crowd work outsourced by the requester using a workflow-based schema. Loosely speaking, the schema contains (i) individual tasks that are considered single units of work, and (ii) basic workflow constructs that encode the interdependency amongst the tasks. Using the task and task dependency specifications in the schema, we are able to find and bind appropriate crowd workers, coordinate their work and request their services to produce data for the crowd work.

- *Work Coordination and Runtime Management*: One of the significant challenges in supporting complex work in crowdsourcing is how to effectively coordinate and monitor the work. We propose a **Coordination Protocol** which is designed to address many of the unique issues in crowdsource platforms for complex work such as discovering a worker, binding a worker to a task and managing and monitoring the quality of work.

In the rest of this chapter, we elaborate on each of them as main design components of our system.

4.1   Encapsulating Worker – Crowd Workers as Services (CWS)

A CWS is a profile documentation of a crowd worker. It consists of a set of data services to derive a comprehensive and flexible view of each worker. Fig. 4.1 illustrates the design of CWS. A crowd worker is depicted in her profile as an entity with exposed service interfaces. Each interface represents an access point to a data service. The data services grant access to not only the personal details of the worker, but also her capabilities, basic profile information, track records and work conditions. We define the following four categories of data services for each crowd worker.
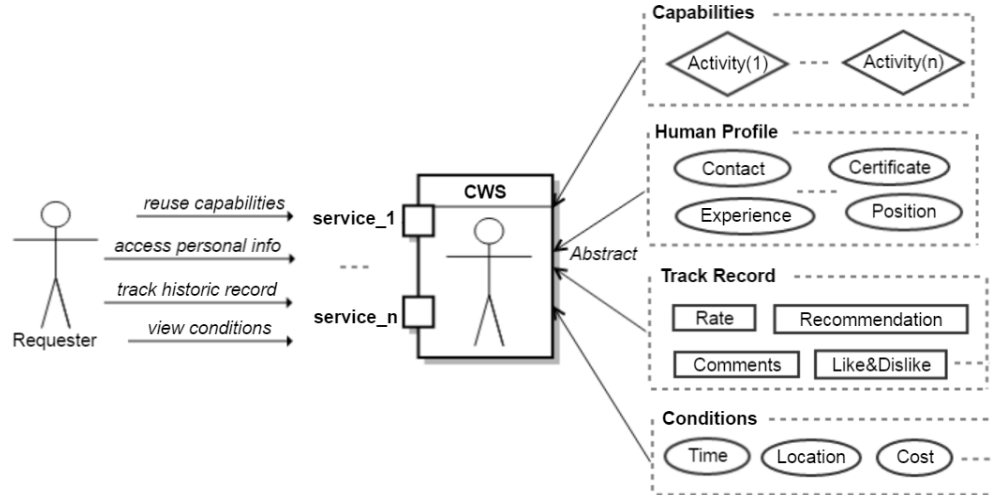
Figure 4.1: CWS Conceptual Design

- **Capabilities Service**: This service returns a list of human activities the worker is capable of performing. Each activity in turn contains the name, input and output data details.

We assume that this information is captured in the system via two ways. First, the worker can declare the types of activities she is able to perform. Second, the system can capture the tasks that were performed by the worker in the past and automatically register them on behalf of the worker. Each task is translated and mapped to the operation-level details of a service definition in a service description language, e.g. WSDL. For example, given an activity (or task) named 'automate-testing' and its input document 'testing specification' and output document 'testing results', we translate it as a service operation named 'automate-testing' which has a request parameter of document type 'testing specification' and a response of document type 'testing results'. This can be encoded concretely in a WSDL document.

What *Capabilities Service* returns in the end is a compiled list of the service operations, which can be used to measure the qualification of a crowd worker.

- **Information Service**: This service returns the basic profile information of the worker. Simply put, the types of information available in this service are those considered relevant to describing a curriculum vitae of a professional (e.g., contact details, position, experiences, qualifications).

  We assume that this information is directly entered and managed by the worker and exposed to the requester via this service.

- **Track Record Service**: This service provides a suite of data relating to the historic performance of the worker. This includes, a historic record of the activities and their approval ratings; the number of activities requested by others in the past; the number of activities rejected by the requester; the number of times the worker was recommended by others, comments on the work (i.e., activities performed), etc.

  The service will provide an aggregated view of the track record, as well as the details of any individual work carried out by the worker.

We assume that this information is collected and managed by the system. These types of information in conjunction with the information service can be used to measure the reputation of a crowd worker.

- **Condition Service**: This service returns a list of conditions or constraints the worker wishes to declare on the system. Some typical examples of the information provided by this service are availability period, preferred contact method, preferred pay level per activity, etc.

  We assume that this information is directly entered and managed by the worker.

One example can be seen in Fig. 4.2. Notionally, from the profile description, we can access capabilities of the service provided by `Shawn` (e.g., automate testing), track records in terms of the list of services that used `Shawn`, as well as other information such as cost and contact details.

```xml
<Participant id="...">
    <name>Shawn</name>
    <gender>Male</gender>
    ...
    <!-- capabilities services -->
    <link title="automateTest"
          href="http://.../automate_testing" />
    ...
    <!-- information services -->
    <link title="getTitle"
          href="http://.../title_getter" />
    ...
    <!-- track record services -->
    <link title="calledServices"
          href="http://.../services_history" />
    <!-- condition services -->
    <link title="pay"
          href="http://.../services_cost" />
    <link title="availability"
          href="http://.../services_availability"
    <link title="contact"
          href="http://.../contact_method" />
</Participant>
```

Figure 4.2: A conceptual view of a crowd worker profile: an example

By identifying and abstracting the various description aspects of a crowd worker (e.g., capabilities, track records) as data services, it is possible to have a uniform

interface-driven access to the available crowd workers and relevant information about them.

More importantly, through modelling a crowd worker as a service (CWS), we can apply the concepts that are well-established in the Service Oriented Computing domain such as service discovery. For instance, we could find the appropriate crowd workers for the appropriate crowd tasks in terms of their qualification and reputation that can be measured via a service quality and discovery technique [Ran03, YWZ$^+$04]. Also, we could coordinate and interact with the crowd workers who are discovered to complete the complex crowd work via a service orchestration and interaction technique [RS05, DS05]. Further, by allowing individuals to autonomously manage their service-oriented profiles, e.g. add or remove a personal service, it provides a way to capture the dynamically changing capabilities and information of crowd workers as their skills and information evolve.

4.2 Encapsulating Work – CCTS (Complex Crowd Task Schema)

The next concept we propose is a model to represent a complex crowd task. It aims to encapsulate the necessary information of a complex task for it to be executed on the crowdsourcing platforms. To this end, we consider the complex task as a piece of *work specification* that describes a goal and constraints. The concept of work specifications is not dissimilar to that of workflow schema or web service composition documents which are common in the workflow and web services coordination techniques.[1]

---

[1]Workflow Patterns: http://www.workflowpatterns.com/patterns/resource/workflow_structure.php

In this work, instead of proposing a new model to represent the specification, we utilise a subset of the existing, well-established workflow concepts and add features that are specific to crowdsourcing (e.g., reward, deadlines). It is noted that our work specification aims to describe a piece of work in a manner that can be fully coordinated by the platform with minimal human intervention during the execution of the work. This means some of the complicated features in the workflow standards such as loops or non-deterministic splits and joins are not considered [VDAVH04].

Overall, in our model, a piece of *work* is described using two concepts: (i) individual tasks where each task represents a unit of work, and (ii) the coordination requirements amongst the individual tasks in order to achieve the goal of the work. In the following, we detail this model further.

4.2.1 Task Definition in CCTS

We consider a complex task as a *work specification*, which is captured as a `Complex Crowd Task Schema` (CCTS). To describe this specification, we first introduce a *task* in our model as a way to describe the range of work to be carried out. A task refers to a *single unit of work* and there are three distinct types of tasks which are named: atomic, composite, and root/goal task. An atomic task is one which has a simple, self-contained definition, meaning it is not described in terms other types of tasks. In the operational sense, this is the only task that is 'executed' or 'carried out by the worker' when initiated. and only one instance of the task executes when it is initiated. A composite task whose implementation is described in terms of atomic tasks. A goal or root task is a unique starting task that contains the descriptions of the goal of the complex work to be carried out. In our work, we assume that a goal task has to be broken down into composite or atomic tasks to have a concrete meaning in the platform.

4. Conceptual Framework for Crowdsourcing Complex Work

All types of tasks share the following definition:

Definition 4.2.1. Let T_{name} be the task name domain, I_{name} be the input name domain and O_{name} the output name domain, C_{name} be the constrain name domain, CV_{value} be the constrain value domain in a crowdsourcing platform. A task t is a tuple, $t = (t_{name}, workload, constraints)$, where:

- t_{name} ($t_{name} \in T_{name}$) denotes the name of the current task. This includes a description and a collection of keywords. t_{name} can be used by the requester to search other similar crowd tasks that have been done in the past, which can help them define the current one.

- **workload** captures the input (i.e., what is given to the task) and output data (i.e., what is produced by the task) of the current task, and is described as $Wrkload = \{Wrk_{input}, Wrk_{output}\}$ where Wrk_{input} consists of a set $\{i_1, ..., i_n | i_i \in I_{name}\}$, Wrk_{output} consists of a set $\{o_1, ..., o_n | o_i \in O_{name}\}$. We assume the input/output data i_i (and o_i respectively) to be an artefact (e.g., documents, zip files) whose names are uniquely identified and addressable within the domains of the crowd platform. This assumption allows us to effectively identify and manage the data generated within the platform.

- **constraints** specify any restriction or condition to be met when performing the current task, e.g. deadline and cost. Constraints is a set of constraint pairs (c_{name}, cv_{name}) where $c_{name} \in C_{name}$ and $cv_{value} \in CV_{value}$.

Figure 4.3 shows an example of our model describing a task. In the example, the task has a name "Write Test cases", the workload contains input document name "`doc: //soc4crowd/write_test_cases/test_plan`" and output document name "`doc: //soc4crowd/write_test_cases/test_cases`". It also specifies a few constraints such as a deadline and cost.
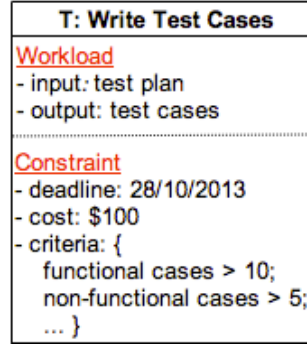
Figure 4.3: An example task

4.2.2 Task Structure in CCTS

Based on the types of tasks we have, to meet the *coordination requirements*, we also define a task structure to provide a skeleton for the crowdsourcing platform to bind appropriate workers, coordinate them, and request their services for the crowd work. A typical structure of a CCTS is shown in Fig. 4.4.
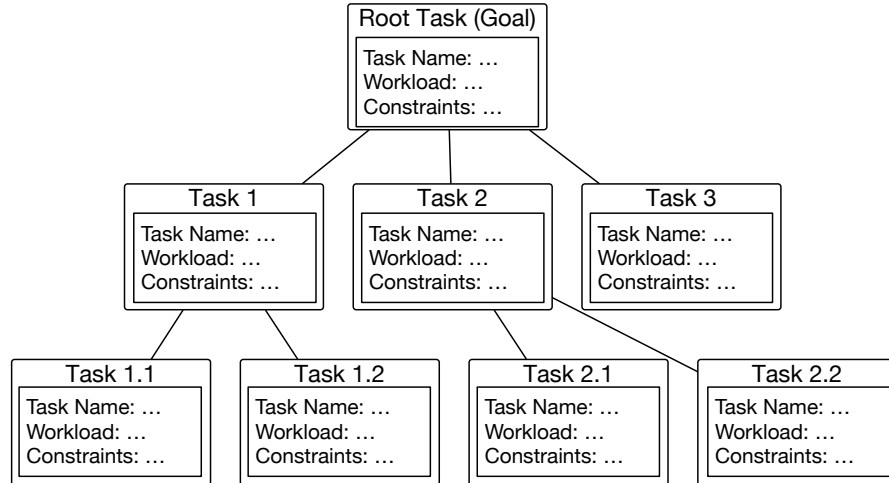
Figure 4.4: CCTS Conceptual Design

It indicates the tree structure of our *CCTS*. Specifically, atomic tasks (i.e., tasks with no children) need to be finished in order to complete the composite task that contains them. The root task represents the original crowd task advertised by the requester; then it can be recursively divided or decomposed into a set of composite tasks until

atomic tasks, based on the above *workload* and *constraint* in task definition; and those atomic tasks are the ones that can be bound with a *CWS* for execution.

When compared with the simple text description used in many of micro-task based crowdsourcing platforms, our *CCTS* can better define the complex crowd task, as its *goal* and *workload* define what work needs to be done while its *constraint* and *structure* define how to get the work done. More importantly, it acts as a schema to get multiple crowd workers involved and coordinate them to complete the complex crowd work, e.g. each atomic *CCTS* is bound with a particular *CWS* and dependencies among those composite or atomic *CCTS* can be used to orchestrate those bound *CWS* to complete the crowd work.

4.3 Coordination Protocol of Crowdsourcing

In this section, we elaborate on a coordination protocol of crowdsourcing in our work to meet the coordination requirement in Sec. 1. When a requester defines his/her work in the schema mentioned above and outsources it to the crowd, first we would find the qualified, trustworthy providers for the requester based on the given schema and providers' profiles. Then, we would get the commitment from both sides on the crowd work performance through their negotiation, and bind each determined provider to a specific task. After that, we orchestrate these bound providers and request their services to complete the outsourced crowd work. To be more specific, this protocol consists of the following phases.

4.3.1 Provider Finding

To find the right provider for the right task in the given schema, we compare the workload and constraint information of a task with providers' information via various services in their profiles. Therefore, we could see if any provider can match the outsourced task in terms of their qualification, motivation, and reputation.

Specifically, first we could see if a provider is capable of doing a task by matching the input and output of this task with any capability service definition in one's profile. Through one's condition services, we could speculate that a provider may be interested in a task if its cost can match his/her required pay, and its deadline is within his/her availability. After that, we may find multiple candidates; at this point we can rank them by considering both service quality metrics and social network metrics, e.g. the average service rate and recommendation from a provider's track record services, the common social connections that a provider shares with this requester via information services, etc. In doing so, we could find and recommend those candidates who are qualified and trustworthy to a certain extent.

4.3.2 Provider Binding

After finding providers, we need to bind each selected one to a specific task before requesting them to produce the data. In other words, we need their commitment on the task performance before its execution, which is represented as an agreement signed by both the requester and bound provider through their negotiation. This agreement determines the final constraint information of a task and contains some detailed terms on task execution (i.e. *if..then* statements), e.g. if a task is finished, then its quality must be measured by a third-party service. Therefore, after binding, this agreement can be used to monitor or supervise the execution, e.g. sending

notification when the deadline is approaching, calling the third-party service for evaluation when a task is done, etc. Further, a provider can be re-bound during the execution through re-negotiation with the requester on some agreement terms, e.g. extending the deadline. At this point, we would check if there is an agreement clash betweem tasks according to their dependencies in the schema, e.g. for two sequential tasks, the extended deadline of the first task becomes later than the second.

4.3.3   Task Execution and Provider Orchestration

When all required providers have been found and bound to specific tasks, it comes to orchestrate them to execute their tasks for the crowdsourced work. Fig. 4.5 shows the lifecycle of each task, during which its execution consists of:

- *service invocation*: this stage is to call the bound provider service to produce data for the task's output. After binding, each provider is automatically requested and their response comes asynchronously (as a human service) to create or update the output. A provider can provide multiple responses to one request for the output update, and the request can be a reminder without providers'response.

- *evaluation*: this stage is to evaluate the task's output when a task is finished by the bound provider. According to the binding agreement, evaluation can be done manually (i.e. by the requester), or automatically (i.e. by our system or the third-party service). Based on the evaluation result and detailed agreement terms, the task would be closed or redone.

As the crowd work is defined as a set of interdependent tasks, the execution of one task may drive another, based on their dependencies. Currently, we consider the interdependencies among tasks as control-flow that can be expressed by an
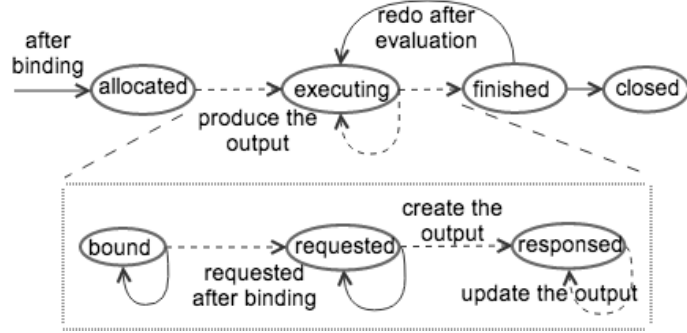
Figure 4.5: Task lifecycle and its execution via service invocation: the above is task lifecycle and the bottom is service invocation for task execution.

executable workflow language. As we represent providers as services, we coordinate the execution of the whole crowd work in the way of orchestrating providers via a workflow engine.

In this chapter, we have introduced the concepts that we consider foundational in supporting complex work crowdsourcing platforms. In proposing those concepts, what we emphasize is to reflect on what Service-Oriented Computing principles can offer in terms of representing the work, coordinating the work and executing the work, and to utilise the existing ideas as much as we can in the platform. This will also help concretising and implementing the ideas using the existing technologies. In the following chapters, we will elaborate on the technical design and implementation of our solution.

Chapter 5

SOC4Crowd Data Model

Up to now, we have discussed the conceptual design and requirements of supporting complex work in crowdsourcing platforms. As of this chapter, we will introduce our complex crowdsourcing system named *SOC4Crowd*. The design and implementation of this system are formed by three main concepts we introduced in the previous Chapter 4 - namely, *CWS*, *CCTS*, and *crowdsourcing coordination protocol*. In realising those concepts, we concretely model and implement a subset of those ideas in Chapter 4 to demonstrate a view - i.e. how employing the existing and mature SOC technologies to the crowdsourcing domain could potentially provide a feasible solution framework.

In this chapter, we introduce the technical design of the data model for *SOC4Crowd*. In particular, we detail one possible way of realising those two key concepts - i.e. *CWS* and *CCTS*, as follows.

5.1 Data Model: CWS

As introduced earlier, CWS allows us to view the crowdsourcing domain from service-oriented perspective. It aims to encapsulate various human capabilities of crowd workers into a common and uniform service interface, thereby abstracting the definition of a piece of work from the actual work (or worker). This also means that, in a similar way of reusing web service, such interfaces can be reused by the requester when (s)he needs to advertise a similar piece of work, but perhaps the work could be carried out by different workers.
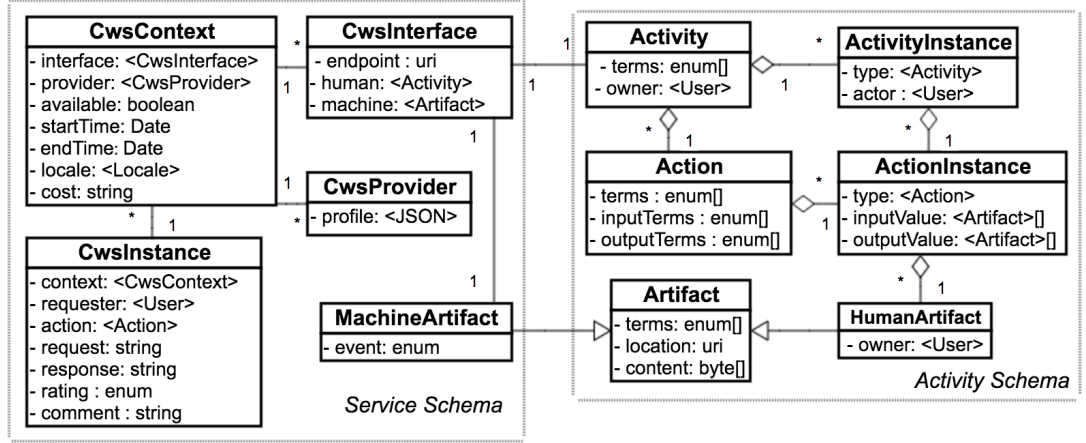
Figure 5.1: CWS Data Model

To realise this fundamental concept, we design it as part of our system data model in Fig. 5.1. Entities are designed to capture the information for the data services, e.g. *capabilities service*, in Sec. 4.1. In particular, those entities can be clustered into the following two categories - i.e. *activity schema* and *service schema*. Next, we will elaborate on each of them.

5.1.1 Activity Schema

As a design to realise the subset of our *Crowd Workers as Services* idea, we mainly focus on its *capabilities service* definition and exposure. To that end, we firstly interpret a *capability* as an *activity*, which is performed by a human actor with a certain amount of time and effort, and whose result produces human artifact.

Then in our vision of human activity in the complex crowdsourcing setting, particularly, we represent it as the one consisting of several atomic actions instead of one single description, and each atomic action is composed of a pair of input and output artifact. For instance, a *software testing* activity consists of two actions - i.e. *manual test* and *automation test*. Both actions accept the same input - e.g. a *test case* document, and produce different output - e.g. *manual test* action generates a *test report* document while *automation test* outputs an executable script.

To realise the above vision, we design our human activity schema as shown in the right-hand side of Fig. 5.1. Specifically, it consists of the following entities.

Artifact and HumanArtifact

`Artifact` entity represents any input or output content that is consumed or produced by either an end user (i.e. a human requester or crowd worker), or a program in our system. With this entity designed, we can proceed and store the result of any program in our system for the crowdsourcing management. More importantly, it is a building block for us to further model human action and activity. In terms of its specific definition, each artifact has the following three common properties, regardless of its actual consumer or producer.

term: this property denotes the unique meaning of an artifact in a specific domain,

e.g. *Test_Plan_Specification* in software quality assurance domain. It is an alternative to the traditional *name* property for the better entity identification. Because its value comes from a list of domain-specific dictionary that is internally maintained within our system, it can avoid the ambiguity. This also means that we assume a simple but extensible taxonomy of terms that can sufficiently represent a domain in which our system operates.

location: this property denotes the web location - i.e. URL, of an artifact.

content: this property denotes the binary content of an artifact.

Regarding the artifact consumed or generated by human - i.e. the requester and crowd workers, we design the `HumanArtifact` entity extending `Artifact` to model the input and output of a human action, e.g. a document, photo, etc. As such, besides those common properties as above, `HumanArtifact` also has its own property - i.e. *owner*: the human owner of current artifact.

With the above `Artifact` and `HumanArtifact` entities defined, we have a building block to further model human activities and actions as their capabilities.

Activity and Action

As introduced earlier, those two entities are designed to represent a type of human activity and its concrete actions that a crowd worker declares to be capable of performing as his or her capability.

In terms of their specific definition, properties composing an `Action` entity are all *terms* of its input and output - i.e. `inputTerm` and `outputTerm`, which are similar to the *term* property of the above `Artifact` entity. That means those input and output terms are domain-specific tags instead of the actual artifact, as the human action is declared but not instantiated yet. In other words, an `Action` entity denotes that, a

human actor is able to perform a certain type of operation with an expected type of input and produce a certain type of output. Under this declaration, when an actual action instance happens, the expected input and output artifact are generated. That will be explained in the later `ActionInstance` entity. As to `Activity` entity here, it is a container grouping multiple `Action` entities to represent a type of human capabilities as a whole.

As exemplified earlier, an `Activity` can be a *Software_Testing* while its nested `Action` can be *Manual_Test* and *Automation_Test*. The `inputTerm` of both *Manual_Test* and *Automation_Test* are *Test_Plan_Specification*, while their `outputTerm` are *Test_Report* and *Test_Script*, respectively.

ActivityInstance and ActionIntance

Those two entities are designed to represent the actual instances of a declared human `Activity` and its nested `Action` entities explained as above, since the same `Activity` and `Action` can be requested to perform multiple times.

As to their specific definition, `ActionIntance` consists of the following properties.

type: this property refers to the type of human action - i.e. the above `Action` entity from which the current instance is derived.

inputValue and *outputValue*: those two properties refer to the actual input and output `HumanArtifact`, which are consumed and produced by the current action instance, respectively.

Likewise, the `ActivityInstance` entity is composed of the following properties.

type: this property refers to the type of human activity - i.e. the above `Activity` entity from which the current instance is derived.

actor: this property represents the human actor who actually performs the current activity instance.

Also similar to the relationship between `Action` and `Activity`, one `ActivityInstance` contains multiple atomic `ActionIntance`.

As we can see from the above human activity schema design, we firstly model human work as artifact consumption and production, given the complex crowdsourcing context. Then we abstract the human work definition from its instances as one's capability for reuse.

5.1.2   Service Schema

Given the above human activity schema, we have introduced a unified way of defining human capabilities in our work. To further abstract it into a service, we design the rest of `CWS` data model entities as follows.

CwsInterface and MachineArtifact

`CwsInterface` entity is designed to represent the interface of a human service. In other words, it specifies what a service can offer at human level, and how it can be accessed at machine level. Specifically, it consists of the following properties.

human: this property refers to a declared `Activity` entity as the service interface at human level.

machine: this property refers to a `MachineArtifact` - i.e. WSDL specification, as the service interface at machine level. Similar to the above `HumanArtifact` entity, here `MachineArtifact` also extends `Artifact` entity to represent any

programming input and output consumed or produced by our system, e.g. WSDL or BPEL artifact. As such, it has its own property - i.e. *event*: the system event that triggers this artifact generation.

endpoint: this property exposes an endpoint address on the web, e.g. URL, for the service access.

Given the above definition, each human service is expressed in the same manner - i.e. what human activity is performed and how it is interacted through the web, regardless of its actual provider.

CwsProvider

This entity represents the actual human provider of the above `CwsInterface`, as there may be multiple crowd workers who are capable of providing the same service. As such, it has a *profile* property in JSON format to describe a particular service provider, e.g. {"name": "John Doe", "contact": "abc@gmail.com", ...}.

CwsContext

This entity is designed to link one specific `CwsInterface` with one particular `CwsProvider` to denote *Who offers What service*. Also, it describes the context of a given CWS for discovery. As such, it consists of the following properties.

interface: this property refers to a `CwsInterface` entity as the service interface.

provider: this property refers to a `CwsProvider` entity as the human provider of the above service *interface*.

available: this property denotes the service availability - i.e. Boolean values as *TRUE* or *FALSE*.

startTime and *endTime*: those two properties denote the service available period, if the above *available* property value is *TRUE*.

locale: this property denotes the language and region of a service, e.g. *en_AU*.

cost: this property denotes the service charge.

Through defining a `CwsContext` entity as above, a human service can be better described with more relevant meta-data, other than its workload information only. Therefore, it can be better matched with a crowd task for interaction.

CwsInstance

This entity represents one specific service interaction instance under a particular context. It is designed to capture some runtime information during the service interaction. Therefore, it consists of the following properties.

context: this property refers to the above `CwsContext` entity to indicate the context in which a service is instantiated.

requester: this property denotes the human service requester.

action: this property refers to `Action` entity from the above human activity model to indicate what exact human action would be taken to perform the requested service.

request and *response*: those two properties are designed to store the raw service request and response messages at runtime.

rating and *comment*: those two properties are designed to store feedback information from the above *requester* after service interaction. As its name suggests, *comment* property is designed to store the requester's comment to illuminate his or her service use experience. Additionally, the requester can also rate a

specific service instance through choosing one of five rating constant that is internally maintained in our system. Those constant are `Poor`, `Average`, `Good`, `Great`, and `Perfect`.

As we can see, those entities designed above express a human service in the WS-* manner, e.g. WSDL and endpoint, for its discovery and interaction. Further, given both *activity schema* and *service schema* in our CWS data model, we provide a way of defining human-readable capabilities and exposing them into machine-readable services for reuse. Next, we will explain how this data model is populated.

5.1.3 Populating CWS

To populate our CWS data model, we introduce the following operations in our system to instantiate and update different entities designed as above.

Activity-to-WSDL Mapping
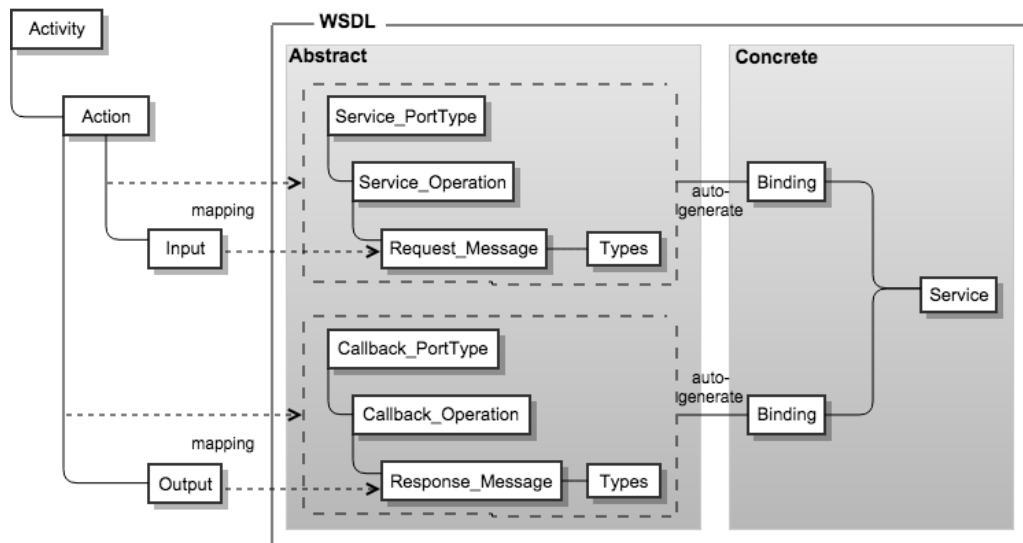
Figure 5.2: Schema Mapping between Human Activity Model and WSDL

To define a human service, we allow each crowd worker to explicitly declare a human activity and its concrete actions that they are capable of performing. As a result, both `Activity` and `Action` are instantiated. To that end, the declaration is done by using those domain-specific *terms* introduced in Sec. 5.1.1. For instance, a software quality assurance analyst can declare his or her software testing activity via *Software_Testing* alike terms; and use *Manual_test*, *Automation_Test* alike terms to define the concrete actions.

With the above declared `Activity` and `Action`, a `CwsInterface` is instantiated at its human level. To make it executable at the machine-level for reuse, we design a schema mapping mechanism to convert the declared human activity and action to a *WSDL* artifact. As a result, a `MachineArtifact` entity will be created and `CwsInterface` will be fully instantiated at both human and machine levels.

As shown in Fig. 5.2, the right-hand side is the structure of *WSDL* schema whose elements are generated based on the left-hand side, the activity model. Particularly, we consider the following two aspects of mapping:

abstract WSDL elements generation: as introduced earlier in Sec. 2.2.2, the abstract part of WSDL defines the web service interface representing what it can do, regardless of how it is implemented. This is similar to our human activity model. Regarding the specific mapping, a WSDL *type* is generated based on the schema of our `Artifact` entity, while the pair-wise WSDL *request message* and *response message* are generated based on the pair-wise human action input and output, respectively. On top of that, each WSDL *operation* and its *portType* wrapper are generated based on a human `Action`. Given the time consuming nature of a human activity, we design its service format in the asynchronous manner. That means each human `Action` generates a pair of *service portType* and *callback portType* for the asynchronous interaction at runtime.

```
<definitions name="software_test" targetNamespace="http://interface.cws.soc4crowd/software_test"
    xmlns="http://schemas.xmlsoap.org/wsdl/" xmlns:xsd="http://www.w3.org/2001/XMLSchema"
    xmlns:wsdl="http://schemas.xmlsoap.org/wsdl/" xmlns:soap="http://schemas.xmlsoap.org/wsdl/soap/"
    xmlns:wsaw="http://www.w3.org/2006/05/addressing/wsdl"
    xmlns:cws="http://soc4crowd/cws"
    xmlns:st="http://interface.cws.soc4crowd/software_test">
    <types>
        <xsd:schema targetNamespace="http://interface.cws.soc4crowd/software_test">
            <xsd:import namespace="http://soc4crowd/cws" schemaLocation="cws.xsd" />
            <xsd:element name="test_plan" type="cws:artifact" />
            <xsd:element name="test_result" type="cws:artifact" />
        </xsd:schema>
    </types>

    <message name="manual_test_request">
        <part name="input" element="st:test_plan" />
    </message>
    <message name="manual_test_response">
        <part name="output" element="st:test_result" />
    </message>

    <portType name="manual_test_pt">
        <operation name="manual_test">
            <input message="st:manual_test_request"/>
        </operation>
    </portType>

    <portType name="manual_test_callback_pt">
        <operation name="manual_test_callback">
            <output message="st:manual_test_response"/>
        </operation>
    </portType>
```

Figure 5.3: Abstract WSDL elements generation

As an example of the above mapping, a *Manual_Test* human `Action` can be mapped into those corresponding WSDL elements in Fig. 5.3, e.g. *manual_test_pt* portType, *manual_test* operation, *manual_test_request* and *manual_test_response* messages, etc.

concrete WSDL elements generation: also as explained in Sec. 2.2.2, the concrete part of WSDL defines how a service is accessed and how it is interacted with a requester, e.g. the request/response message format and its transport protocol over the network. Given the mature service interaction related technologies, we auto-generate those concrete elements. Specifically, a WSDL *binding* is auto-generated using *WS-Addressing/SOAP* protocol, while a *service* is generated accordingly with a default endpoint given by our system.

As a result of the above *Activity-to-WSDL* mapping mechanism, we instantiate a `MachineArtifact` entity with the generated WSDL and associate it with `CwsInterface`. With this machine-level interface, we can later reuse the associated human capa-

```xml
<?xml version="1.0" encoding="UTF-8"?>
<soap:Envelope xmlns:soap="http://www.w3.org/2001/12/soap-envelope"
    soap:encodingStyle="http://www.w3.org/2001/12/soap-encoding"
    xmlns:wsa="http://schemas.xmlsoap.org/ws/2004/08/addressing"
    xmlns:xsd="http://www.w3.org/2000/10/XMLSchema"
    xmlns:cws="http://soc4crowd.org/cws"
    xmlns:sof="http://interface.cws.soc4crowd/software_test">
    <soap:Header>
        <wsa:To>http://localhost:8888/manual_test_binding</wsa:To>
        <wsa:Action>http://localhost:8888/manual_test</wsa:Action>
        <wsa:MessageID>uuid:8f2d82f5-2342-4234-asdf-asdas2ds3sd</wsa:MessageID>
    </soap:Header>
    <soap:Body>
        <sof:manual_test_request>
            <sof:test_plan>
                <cws:artifactTerm>Test_Plan</cws:artifactTerm>
                <cws:artifactLocation>
                    https://docs.google.com/document/d/1zJkF5I5enTtEyl07sP14eeDJGA81WNjCfRN0qAqpgmU
                </cws:artifactLocation>
            </sof:test_plan>
        </sof:manual_test_request>
    </soap:Body>
</soap:Envelope>
```

Figure 5.4: Human capability reuse through SOAP

bilities through the service discovery, invocation and interaction. For example, when the above *Manual_Test* human action has been exposed as a service, a human requester can use her client application, which may not know our internal human activity model, to send a service request according to the mapped WSDL definition. As shown in Fig. 5.4, the input of *Manual_Test* human action, which is *Test_Plan* document, is encapsulated into a SOAP request message - i.e. the `<sof:manual_test_request>` XML fragment. Then the client application used by the crowd workers receives and parses this request message to extract the input for the actual service provider to perform the *Manual_Test* human action. When the actual work is finished with an output, the client application constructs a SOAP message containing that output and sends it to that requester as a service response. In doing so, this software testing human capability is reused in a distributed environment.

CWS Publication and CWS Interactions

After defining a human service through instantiating the entities defined in the CWS data model, a crowd worker needs to publish the content for discovery. To

62

complete the description of a human service, we allow the crowd worker to upload some profile information and describe other service context, e.g. availability, start and end time, location, cost, etc. As a result, both `CwsProvider` and `CwsContext` entities in the CWS data model are populated, and ultimately are associated with the `CwsInterface`.

Other aspects of the CWS model cover the runtime issues. When a CWS has been discovered and requested (i.e., the service is now bound to the human worker behind it), we process this CWS interaction at both machine (software) and human levels. At machine level, we instantiate a `CwsInstance` entity to record the interaction details, e.g. the actual human requester, the request and response messages, the comment and rating feedback, etc. At human level, we instantiate `ActivityInstance`, `ActionInstance`, and `HumanArtifact` entities with the information from `CwsInstance` and visualize them on the dashboard of crowd workers. Therefore, they can view the request details and retrieve their human action input. And subsequently, they can perform the actual action to produce the output.

5.2   Data Model: CCTS

To meet the coordination requirements in Chapter 1, we propose a data model named Complex Crowd Task as a Schema (CCTS). We simply refer to this as CCTS schema or CCTS. The information and structure in CCTS provide a skeleton for our *SOC4Crowd* system to find and bind appropriate workers, coordinate them, and request their services for the advertised task.

5.2.1   CCTS Schema

Similar to the above CWS, we also design it as part of our system data model and its schema can be seen in Fig. 5.5. Next, we will elaborate on those entities in this schema from the following perspectives - i.e. task definition, task structure, and task context.
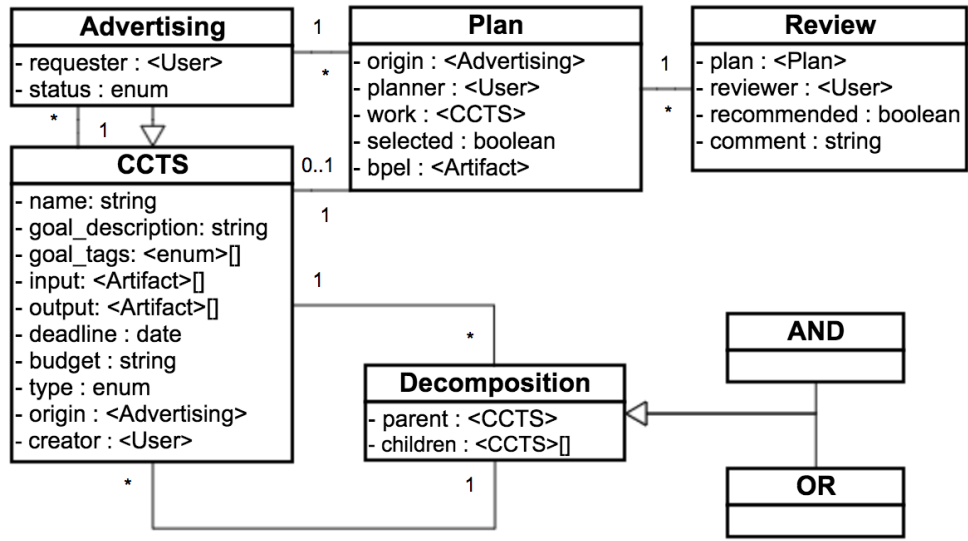
Figure 5.5: CCTS Data Model Schema

Task Definition – the entity CCTS

As explained earlier in Sec. 4.2.1, one essential aspect of our CCTS concept is to define the crowd task that instructs *what needs to be done*. Therefore, it has been designed to consist of those information - i.e. task *name* or *goal*, *workload*, and *constraints*. Correspondingly, we design the `CCTS` entity in Fig. 5.5 and elaborate on its detailed attributes as follows to encapsulate those information.

Goal of a crowd task is designed to consist of a `goal_description` and a collection of `goal_tags` attributes. The former is a human-readable attribute and its

value can be read by any crowd worker who is interested in this task, while the latter is a machine-readable attribute and can be used to search other similar tasks done in history, which in turn helps to define the current one, and also can be used to match a CWS for binding and execution.

Workload of a crowd task is designed as pairwise `input` and `output` attributes. They are both referring to a collection of `Artifact` entities, each of which consists of a `tag` and `location` attributes, according to our earlier CWS design.

Constraint of a crowd task is designed as `deadline` and `budget` attributes. As their names suggest, they are used to capture the time and cost constraints of completing a task.

Besides those essential attributes of defining a task as above, our CCTS entity also has a `type` attribute to denote the granularity of crowd work in the defined task, i.e. *atomic*, *composite*, or *root*. Specifically, an atomic CCTS encapsulates a single unit of the above workload information and cannot be further decomposed or divided, e.g. a single input and output artifact. Moreover, an atomic CCTS in our system should be able to be bound with a CWS instance for execution. By contrast, a composite CCTS should be further decomposed into a set of atomic CCTS for more meaningful and feasible execution through the proper workload division, while the root CCTS represents the initial crowd task.

Task Structure – entities Decomposition, AND, OR

Besides the task definition, another important aspect of our CCTS concept is to represent the work structure, which indicates *how to complete a task*. As discussed earlier in Sec. 4.2.2 and from the above `type` attribute design, our CCTS has a decomposable tree structure based on its *workload* and *constraint*. Within this

structure, its root CCTS represents the original crowd task, while its atomic sub-CCTS represents the single unit of work that needs to be done in order to complete the original task. To realise this structure, we design the following entities.

Decomposition entity is designed to manifest the tree structure of our CCTS data model. As such, it consists of the `parent` and `children` attributes referring to a given CCTS and a set of its sub-CCTS after decomposition, respectively.

AND and **OR** entities represent the specific type of CCTS decomposition. Therefore they extend the above `Decomposition` entity to not only manifest the tree structure, but also to maintain the data flow among `children` CCTS and their `parent`.

Through the above task definition and structure entities, we can have a clearer description of *what needs to be done* and *how to get it done*, when compared with the simple task description in most of micro-task based crowdsourcing platforms.

Task Context – entities Advertising, Plan, Review

The following entities further add information to the tasks.

Advertising entity represents the original crowd task advertised by the requester. Therefore, it extends the above `CCTS` entity to have the same task schema. Also, it has its own advertising context attributes - i.e. i). `requester`: the human who advertises his or her task to the crowd; ii). `status`: the processing status of an advertised crowd task, e.g. *advertised*, *planning*, *executing*, etc.

Plan entity is designed to represent the result of task decomposition in our complex crowdsourcing context. We will explain the task decomposition later in the next chapter. Here in general, a `Plan` entity manifests the context - i.e. *who*

has done *what planning work* on *which advertised task*. In particular, it consists of the following attributes - i.e. i). `origin`: the originally advertised crowd task referring to the above `Advertising` entity; ii). `planner`: the human who has done the task decomposition and submitted the current plan; iii). `work`: the actual planning work whose result is a fully structured `CCTS` entity; iv). `selected`: a flag indicating if the current plan is selected; v). `bpel`: a WS-BPEL artifact as the result of a selected plan.

Review entity is designed to capture the feedback on the above task `Plan`. As such, it consists of `recommended` and `comment` whose value come from the human `reviewer`.

Those entities above manifest a task in our complex crowdsourcing context, particularly the task planning phase of our crowdsourcing lifecycle. We will briefly explain that in the later CCTS model population section.

5.2.2   Populating CCTS

CCTS model underpins our approach for creating and managing the complex work in the framework. Therefore, populating this model involves explaining the operations that are involved in various stages of the lifecycle of the complex work represented in our system. Each stage of the lifecycle is designed to populate a specific part of our CCTS data model. When the model is fully instantiated, it can be bound with our CWS instances together for a crowdsourcing process execution. To properly manage those operations, e.g. their participants, sequence, and result, we design a two-phased crowdsourcing lifecycle management as the operational model of the framework. We will elaborate on the design in the next chapter.

Chapter 6

SOC4Crowd Operation Model

In this chapter, we specify the design of our operation model, namely *Two-Phase Crowdsourcing Management*. In detailing the operational issues of the framework, we explain how the data models `CWS` and `CCTS` are utilised to support the complex task crowdsourcing.

6.1 Overview

SOC4Crowd framework divides the lifecycle of complex crowdsourcing into two phases: *Plan* and *Execution*. As shown in Fig. 6.1, during the plan phase, the system allows the requester to use "crowdsourcing" to generate a decomposed working plan of the advertised task. From this plan, the system will generate an instance of the *CCTS* data model. Then in the execution phase, the system allows the requester to "crowdsource" the execution of the decomposed working plan via orchestration of the bound human actions and necessary machine-level communications. The detailed design of each phase will be discussed in Sec. 6.2 and Sec. 6.3, respectively,

here we present a quick overview on each phase as follows.
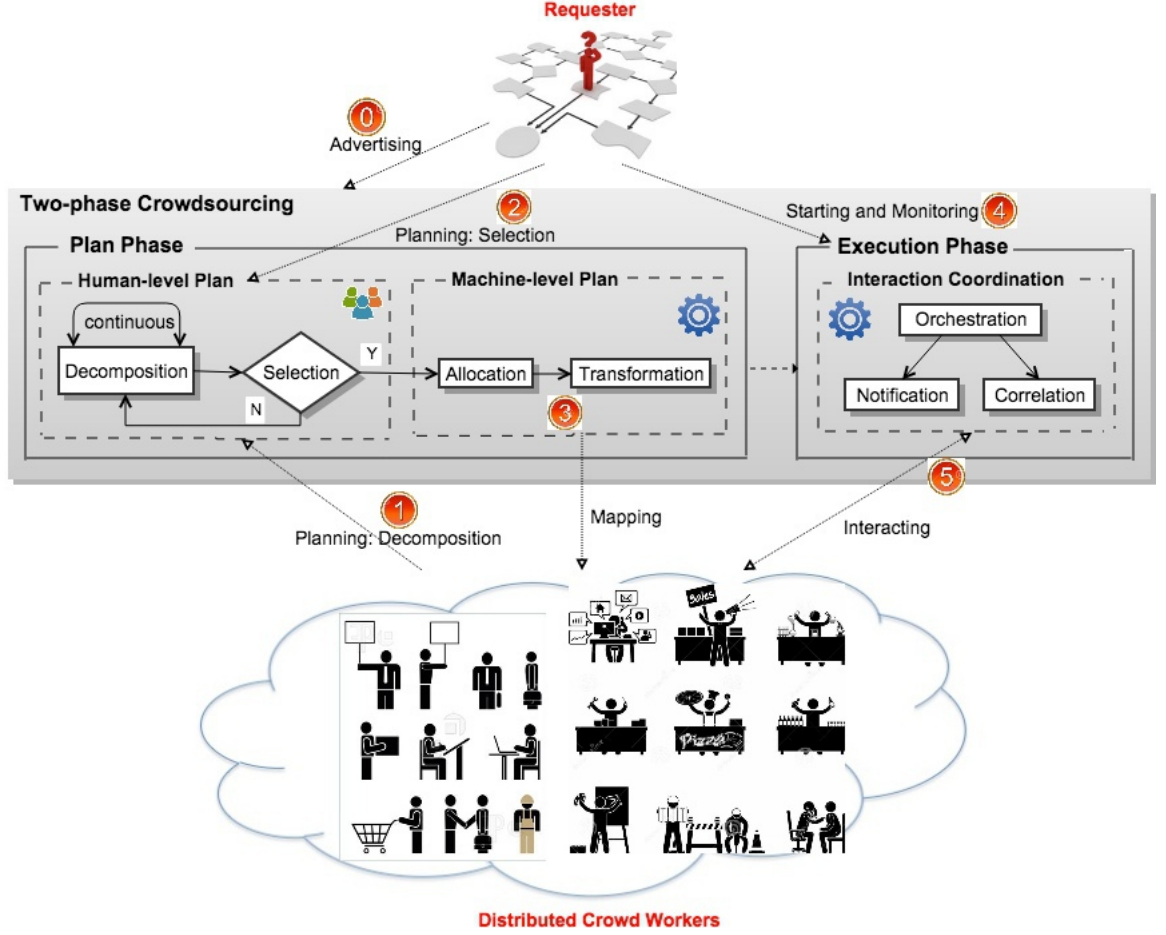
Figure 6.1: Two-Phase Crowdsourcing Framework Overview

6.1.1   Crowdsourcing Work Plan

Plan Phase is a new stage that we add into the crowdsourcing lifecycle. It is designed to detail the advertised complex task definition and generate a ''workable' plan by decomposing the complex task down into sub tasks that can be directly bound to known services in the framework. The **Plan Phase** idea is inspired by the *Crowd Design* introduced in Chapter 3, which utilises the crowd wisdom to create a high-quality design artifact. Similarly, the task plan generation is also a

creative process wherein we can utilise the crowd to brainstorm multiple candidate plans; then vote and choose the 'best' as the final working plan. In other words, we crowdsource the task plan before its execution.

To this end, we firstly ask the requester to work with the crowd together to generate a human-readable plan. Then we transform it into a machine-readable schema for the auto-coordination of crowd workers in the later execution phase.

Human-level Plan Generation

To start with, the requester advertises his/her complex task through our framework, which is denoted as step 0 in Fig. 6.1. As a result, our CCTS data model gets initialized. Particularly, an `Advertising` entity is instantiated as a root `CCTS` for the following task plan steps at human level.

Task Decomposition. To generate a work plan on the advertised task, as shown by step 1 in Fig. 6.1, each distributed crowd worker can participate in this plan phase through their own task decomposition. Those crowd participants can use the provided decomposition `Operator` - i.e. `AND` and `OR`, to continuously divide the `root CCTS` into a set of `atomic sub-CCTS` and populate each of them. As a result, each participant contributes a fully-structured `CCTS` instance as their own work plan. Then they can submit it to instantiate a `Plan` entity for *Selection* as follows.

Plan Selection. To guide the later crowd task execution, the requester needs to select one final plan from those candidates submitted by the crowd, which is denoted by step 2 in Fig. 6.1. During the selection, each submitted plan becomes read-only and it can be commented and recommended by any other crowd workers. As such, multiple instances of *Review* entity are created and associated with each plan candidate to help the requester make a final decision.

After going through the reviews, the requester can select one of submissions as the final work plan; or (s)he can start the planning phase over if none of them is acceptable.

Through the above steps, the originally advertised task gets detailed and refined with structure and sub-tasks. Also, thanks to the joint participation in task plan generation, both the requester and crowd workers are on the same page, in terms of *what needs to be done* and *how to get it done*, which would in turn increase the crowdsourcing quality.

Machine-level Plan Transformation

As a result of the above human-level plan generation, we have a fully instantiated CCTS data model. To make it executable at machine level, we need to get multiple `CWS` involved and generate a service orchestration specification for the later auto-coordination. To this end, we auto-allocate each `atomic sub-CCTS` in the above human plan to an appropriate `CWS`; then we auto-transform that fully instantiated CCTS schema to a web service orchestration schema - i.e. BPEL, in the following steps.

Allocation. As denoted by step 3 in Fig. 6.1, we try to auto-map an `atomic sub-CCTS` to an appropriate `CWS` in the distributed crowd. To realise that, we design a *matching-filtering-ranking* pipeline as follows.

Matching: we try to see if a crowd worker is capable of performing an atomic CCTS by matching its goal and workload with his/her CWS interface, e.g. human activity and actions.

Filtering: as a result of the previous *matching* pipe, we may find multiple qualified candidates. Here we can filter some of them by considering their CWS context and the constraint of current CCTS, e.g. we can see if their service cost can be covered by the budget of current CCTS, and if their service availability is no later than the deadline of current CCTS, etc.

Ranking: as a result of the previous *filtering* pipe, we may still have multiple candidates remaining. Here we rank those candidates and pick up the top one. To do so, we take into account their personal profile information, e.g. experience, certificate, etc., and service history, e.g. the average service rates and recommendation times from the track record.

The detailed design of the above pipeline can be seen in Sec. 6.2.2. As a result, we can find and bind a particular CWS, which is qualified and trustworthy to a certain extent, with a particular atomic CCTS.

Transformation. With the bound CWS after the above *Allocation* step, our CCTS schema is then auto-transformed into an executable service orchestration schema (i.e. BPEL) for the auto-coordination in the later execution phase. To realise that, we design a schema mapping mechanism as follows.

Elements mapping: for instance, we map an `atomic CCTS` element to a pair of `invoke` and `receive` service elements in BPEL.

Dependencies mapping: we map the data-flow dependencies in CCTS to the control-flow dependencies in BPEL.

The detailed design of the above transformation mechanism can be seen in Sec. 6.2.2. As a result, we generate a BPEL artifact for the later auto-coordination at runtime.

Given the complex crowd task instead of the micro-task, we have designed a separate **Plan Phase** in the crowdsourcing lifecycle as above to clearly define *what needs to be done*, *how is it going to be done*, and *who will do what*, before the crowd task execution. Furthermore, we make the planning result executable to bring the auto-coordination support from service orchestration perspective.

6.1.2 Crowdsourcing Work Execution

When compared with the traditional crowd work execution, we bring the auto-coordination of multiple crowd workers into our framework, given the crowdsourcing work plan result as above. As shown by step 4 in Fig. 6.1, the requester needs to manually kick off the crowdsourcing execution through sending an initial service request to our framework. Since then, we coordinate those allocated crowd workers automatically through executing the service orchestration schema, i.e. BPEL, which is generated from the above plan phase. As shown by step 5 in Fig. 6.1, this auto-coordination is reflected as managing interactions at both human and machine levels.

Human-level Interactions are realised as both online and offline **Notification** in our framework. Specifically, each crowd worker gets notified by both an online dashboard message and an offline email, when the input of their allocated CCTS is ready.

Machine-level Interactions are realised as the asynchronous message **Correlation** and service **Orchestration**. Specifically, when each individual worker finishes their allocated CCTS, the late response from that individual CWS is asynchronously correlated to its earlier request; meanwhile, a global service orchestration is enabled to manage dependencies and route messages among multiple CWS while running the orchestration schema (i.e. BPEL).

Being inspired by the workflow perspective from SOC domain, we design the above **two-phase lifecycle** as our system operation model for the complex work crowdsourcing management. Specifically, being motivated by the crowd design pattern, we add a separate **plan phase** to crowdsource the work plan of an advertised complex task. Then we bring the auto-coordination of multiple crowd workers into the crowdsourcing **execution phase**, with a provided coordination schema from the earlier plan phase.

Architecture Overview

Here we introduce an overview of our system architecture whose modules are the participants of our two-phase crowdsourcing lifecycle as above. As shown in Fig. 6.2, this high-level architecture consists of three key modules - i.e. *RequesterClient*, *CwsClient*, and *CoordinationMiddleware*.
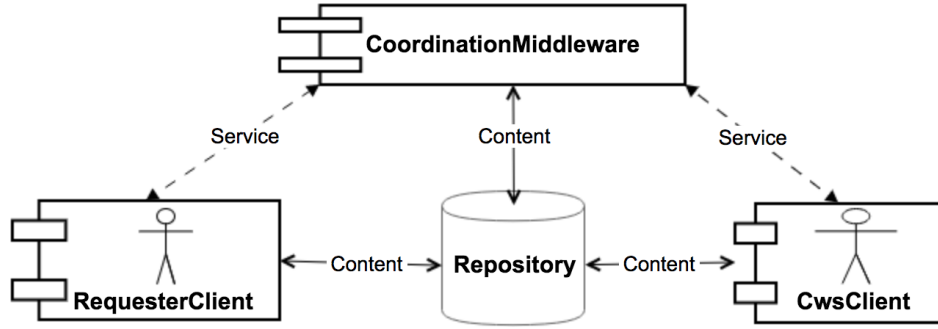

Figure 6.2: High-level Architecture of SOC4Crowd framework

RequesterClient is an end-user application that helps a human requester participate in each crowdsourcing phase, e.g. advertising a crowd task; reviewing and selecting a crowdsourcing plan; starting and monitoring the crowdsourcing execution, etc.

74

CwsClient is an end-user application that allows each crowd worker to create and publish their human services. Also, it facilitates their participation in each crowdsourcing phase, e.g. conducting task decomposition during the plan phase; handling their human service request and response during the execution phase, etc.

CoordinationMiddleware is an intermediary between the above applications to coordinate their interactions and manage our two-phase crowdsourcing lifecycle as a whole. For instance, it broadcasts a new crowd task advertised by the requester to the crowd; it transforms the selected `CCTS` and `Plan` instance to a BPEL specification during the plan phase; it deploys and starts a BPEL process to orchestrate and interact with `CWS` and requester during the execution phase; etc.

With the above participants, we can later elaborate on the design of each step in our operation model in Sec. 6.2 and Sec. 6.3. Then we will detail the design and implementation of the above overall architecture in Chapter 7.

Operation Sequence

To better illustrate the design of each specific step of our operation model in the following sections, here we introduce some basic elements of *UML sequence diagram* as our operation sequence notation. As shown in Fig. 6.3, there are four types of elements as follows.

Actor element represents the end user who kicks off a specific sequence. In our case, it represents either a human requester or crowd worker.

Instance/Object element represents an instance or object of the participating system module. In our case, it is an instance of either *RequesterClient, CwsClient,*
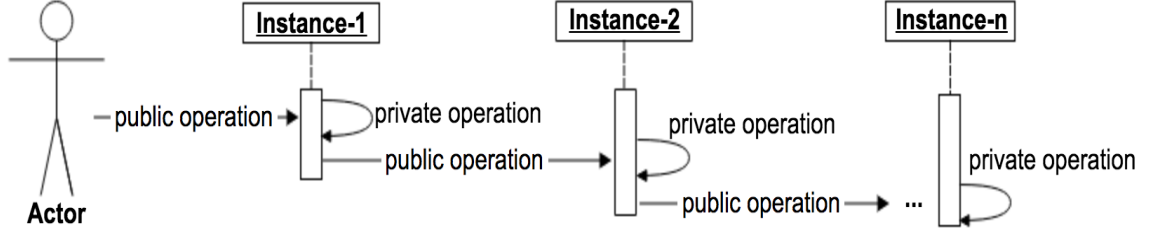
Figure 6.3: UML Sequence Diagram Notation

or *CoordinationMiddleware* applications;

Operation element is a specific *service* or *method* provided by one particular `Instance` as above, which can be requested by another. It represents the atomic interaction between those instances. In our case, it is indicated by the arrow notation and its right-hand side object is the operation provider while its left-hand side object is the operation caller. Moreover, it can be either a *public operation* invoked by another object, or a *private operation* called internally.

Activation Timeline element shows what happens in sequence (i.e. from top to bottom) within a particular object when its provided `Operation` is activated.

Through those elements as our design notation, we can clearly illustrate the design of a specific step in our operation model from the sequential or timeline perspective.

6.2 Plan Phase – Populating and Transforming CCTS Model

As discussed earlier, most of micro-task based crowdsourcing platforms define the crowd work via a simple task description which, though, is not clear enough for the complex work. Therefore, we design a separate `Plan Phase` to crowdsource a plan of the complex crowd task before its execution. As introduced in Sec. 6.1.1, our

plan phase consists of the following two sub-stages - i.e. *human-level planning* and *machine-level planning*.

6.2.1 Human-level Planning: CCTS Population

At this stage, both the requester and crowd work together on the complex task definition and refinement. As a result, a fully structured `CCTS` instance is populated as a task `Plan` before its execution. As explained in Sec. 6.1, the current stage consists of the following three concrete steps - i.e. `Advertising`, `Decomposition`, and `Selection`, in order.

Advertising

This is an initial step to start the whole planning phase - i.e. the human requester advertises his/her original complex task to the crowd. We design its specific sequence in Fig. 6.4.
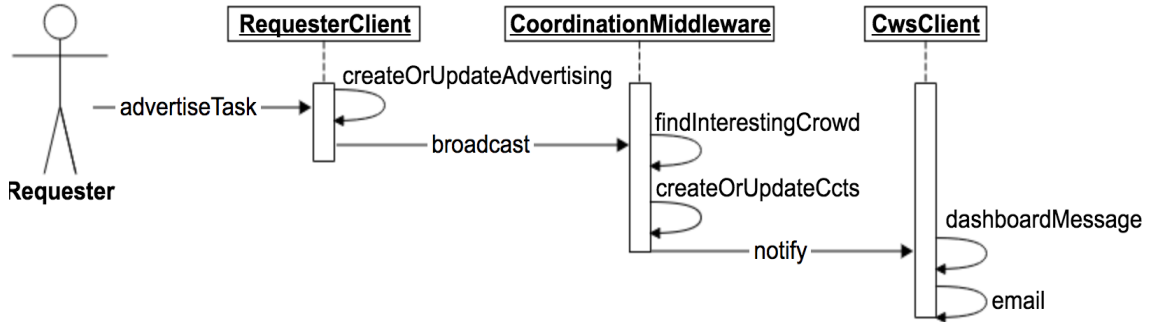

Figure 6.4: Advertising sequence at human-level planning stage

As illustrated, the *Advertising sequence* is kicked off by the human requester who triggers the public `advertiseTask` service of `RequesterClient` application when (s)he posts his/her task. Then the private `createOrUpdateAdvertising` operation

of `RequesterClient` is called internally to create an instance of `Advertising` entity in our data model. After that, it invokes the public `broadcast` operation of `CoordinationMiddleware` to broadcast this new advertising to the crowd.

Subsequently, `CoordinationMiddleware` calls its internal `findInterestingCrowd` operation to target at a certain group of crowd workers who may be interested in the new advertised task. To realise that, it tries to match their human `Activity` and `Action` with the current `Advertising` in our data model. Particularly, it tries to see if the `term` attribute of any `Activity` or `Action` entity is matched with the `goal_tag` attribute of `Advertising` entity. After finding a group of those crowd workers, our `CoordinationMiddleware` then calls its private `createOrUpdateCcts` operation to create a root `CCTS` instance for each of them by copying the information of new created `Advertising`. In doing so, each of them can have a crowd task copy on their dashboard for their own task decomposition later on.

Lastly, our `CoordinationMiddleware` asks `CwsClient` to inform those matched crowd workers about this new advertising, through invoking its public `notify` operation. Internally, our `CwsClient` realise this notification through both `dashboardMessage` and `email`. As notified, they can proceed the follow-up task decomposition.

Decomposition

At this step, each notified crowd worker from above is doing their own planning work through continuously decomposing a root `CCTS`, which represents the original `Advertising`, into a set of atomic sub-CCTS. We design its specific sequence in Fig. 6.5.

As illustrated, the *Decomposition sequence* is kicked off by the crowd worker who triggers the public `decomposeCcts` service of `CwsClient` application. Then inter-

nally, `CwsClient` calls its private `createOrUpdateCcts` operation to create multiple `CCTS` instances and specify their relationships in our data model. The current crowd worker can continuously invoke that public `decomposeCcts` service to generate a fully structured CCTS instance as his/her own plan.
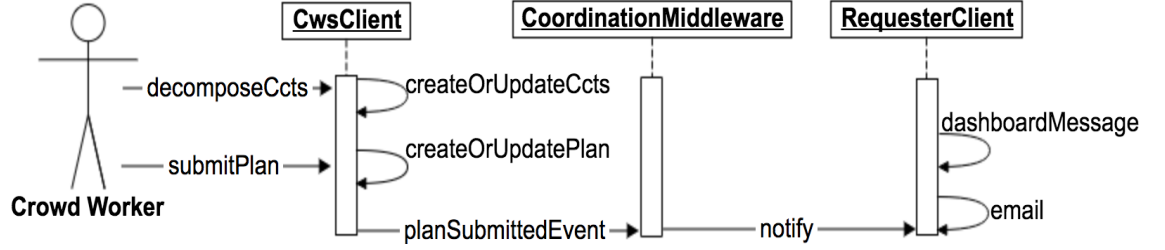
Figure 6.5: Decomposition sequence at human-level planning stage

Once (s)he finishes his/her planning work and submits it through the public `submitPlan` service, `CwsClient` internally creates an instance of `Plan` entity in our data model through its private `createOrUpdatePlan` operation. Then it fires a `PLAN_SUBMITTED` event to `CoordinationMiddleware` who subsequently asks `RequesterClient` to `notify` its end user - i.e. human requester, about a new plan submission.

Selection

At this step, the requester can start a poll to get reviews on each plan submission from the crowd; then select one of them as the final work plan. We design its specific sequence in Fig. 6.6.

As illustrated, the *Selection sequence* is kicked off by the requester who triggers the public `startSelection` operation of `RequesterClient` when (s)he has received enough plans from the crowd and decided to choose a final one from those candidates. Internally, our `RequesterClient` calls its private `createOrUpdateAdvertising` operation to update the `status` of original advertised crowd task from `planning` to `selecting`. As such, crowd workers are not allowed to continue their plan work or

79

submission, as `CwsClient` disables the above `Decomposition` sequence due to the task status transition.
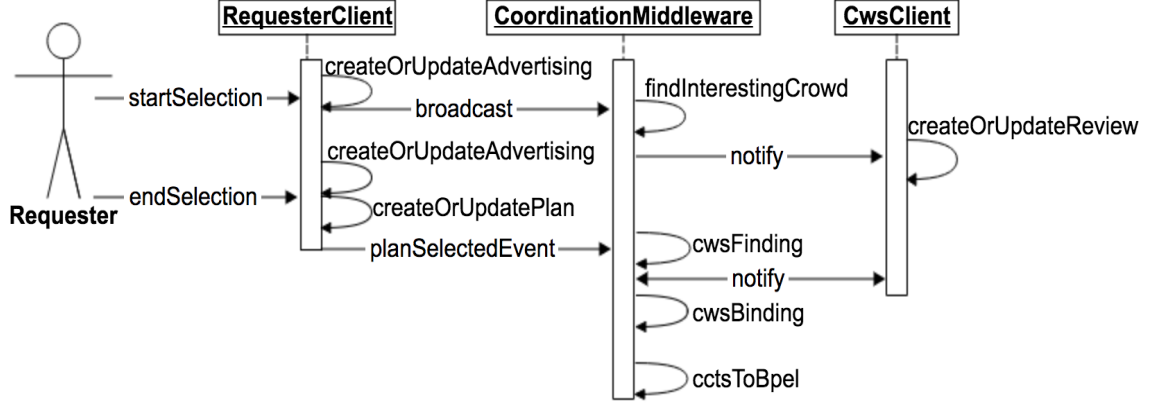
Figure 6.6: Selection sequence at human-level planning stage

Then our `RequesterClient` asks `CoordinationMiddleware` to broadcast this plan selection to the crowd for voting and commenting through its public `broadcast` operation. To this end, `CoordinationMiddleware` calls its private `findInterestingCrowd` operation again to find another group of crowd workers excluding those who have done the above `Decomposition` sequence for voting. After that, it asks `CwsClient` to inform those crowd workers about the plan voting through the public `notify` operation. Then the notified crowd workers can give the feedback to any of the submitted plans through the `createOrUpdateReview` operation that instantiates `Review` entity in our data model.

Once the requester has gone through all reviews from the crowd, (s)he can determine the final plan through invoking the public `endSelection` operation of `RequesterClient`. Again, `RequesterClient` calls its `createOrUpdateAdvertising` operation internally to update the `status` of original advertising from `selecting` to `allocating`. As a result, crowd workers are not allowed to continue their voting, as `CwsClient` disables the review creation due to the task status change. After that, `RequesterClient` updates the `selected` attribute of the chosen `Plan` entity as `true` through invoking its private `createOrUpdatePlan` operation.

Lastly, it fires a `PLAN_SELECTED` event to `CoordinationMiddleware` who subsequently starts the machine-level planning process whose details will be explained in the follow-up Sec. 6.2.2.

Through the above detailed sequence design, we demonstrate how our *SOC4Crowd* framework realises each step of the *Human-level Planning* stage through interactions among those key modules or participants, i.e. *CwsClient*, *RequesterClient*, and *CoordinationMiddleware*, of our system architecture depicted in Fig. 6.2. As a result of human-level planning stage, we have initialized our CCTS data model, e.g. `Advertising`, `CCTS`, and `Plan` entities, which in turn are the input of the following machine-level planning stage and execution phase.

6.2.2 Machine-level Planning: CCTS-to-BPEL Transformation

In order to auto-coordinate multiple crowd workers in the later execution phase through the above human plan, we need to bind it with the appropriate `CWS` and make it executable. To this end, we design an underlying machine-level planning stage to auto-allocate each atomic `CCTS` to an appropriate `CWS`; then auto-transform the above human plan result into a service orchestration specification - i.e. BPEL.

Allocation

As recalled from the above `Selection` sequence design, when a final work plan has been selected by the requester, our `CoordinationMiddleware` kicks off the current machine-level planning process through finding and binding CWS with CCTS - i.e. its internal `cwsFinding` and `cwsBinding` operations in Fig. 6.6, respectively.

To start with `cwsFinding`, this operation realises that `matching-filtering-ranking` pipeline design in Sec. 6.1.1 for CWS discovery as follows.

Matching Pipe: This pipe aims to find those crowd workers who are capable of performing an atomic CCTS. To this end, we match its `goal` and `workload` information with one specific human `Action` of their `CWS Interface`.

Specifically, we try to see if:

there is any match between `goal_tags` attribute of current CCTS entity and `terms` attribute of `Action` entity in our data model.

the `input_tags` of CCTS contains all `input_terms` of a CWS, and if the `output_terms` of CWS contains all `output_tags` of current CCTS. In doing so, we can make sure that the crowd worker will have enough input to conduct his/her human action and its output will be sufficient to meet the task requirement.

As per the pipeline design, the matching result will be passed as the input to the follow-up `Filtering` pipe, when multiple qualified CWS are found.

Filtering Pipe: This pipe aims to filter those crowd workers who are not available or suitable to take a given CCTS, even though they are capable of conducting it. To this end, we compare the `constraint` information of current CCTS with the `context` of a particular CWS.

Specifically, we try to see if:

the `deadline` attribute of current CCTS entity is no later than the `availability` attribute of a CWS.

the `budget` of current CCTS is greater or equal to the `cost` of a CWS.

As a result, we can find those crowd workers who are not only capable of performing the given task, but also able to finish it in time and on budget. Likewise, this result will be the input of the follow-up `Ranking` pipe, when there are still multiple candidate CWS remaining.

Ranking Pipe: This final pipe aims to order those candidate CWS who ''survived'' from the above `Matching` and `Filtering` and returns the top one as the result of `cwsFinding` operation. To this end, we try to measure the human service quality from the historic perspective and use it as the ranking criteria.

Specifically, we calculate the average `rating` of each candidate CWS and rank them in a descending order. The `rating` attribute of CWS entity is a built-in numeric constant or enumeration in our system, which is used by the requester to evaluate each CWS instance after their interaction is finished. A typical example can be: {''Poor'':1-3}, {''Average'':4-6}, {''Good'':7-8}, {''Great'':9-10}. After that, we return the first ranking result as the above `cwsFinding` operation output.

Now we need to bind the found CWS with its CCTS in order to ensure the proper data flow from the actual service to the required task. To this end, another operation - i.e. `cwsBinding`, is called by `CoordinationMiddleware` internally to instantiate the `CwsBinding` entity as a link between CWS and CCTS in our data model, which records *who* will perform *what action* for *what task*. In doing so, during the later execution phase, we can trace and manage the data-flow between CWS and CCTS, e.g. we can notify or chase the crowd worker up to perform their action to produce output for their bound CCTS when the input is ready; and then we can populate the output of CCTS when it is generated in the CWS response.

Transformation

With the bound CWS in our human-level planning result - i.e. CCTS schema, after the above `Allocation`, we need further to make it executable in order to auto-coordinate those involved crowd workers in the later execution phase. To this end, we transform our human-readable plan to a machine-readable service orchestration schema - i.e. BPEL, considering the web service format of CWS.

The transformation process is managed by `CoordinationMiddleware` via its private `cctsToBpel` operation at the end of Fig. 6.6. Internally, this operation realises a mapping algorithm in Algo. 1 for BPEL generation. As shown in the pseudo code, our `CCTS-to-BPEL` algorithm is realised through the following three functions - i.e. `Transform`, `Template`, and `Traversal`.

Transform Function: This function is the starting point of our algorithm and it aims to outline the transformation sequence. To start with its signature, it takes the first argument - i.e. the selected human-level `Plan` entity, as its input, while it generates a BPEL `Artifact` for its second argument as its output. Within its function body,

Line 2: we retrieve the fully structured `CCTS` instance from the first function argument - i.e. `Plan` entity;

Line 3: we call the follow-up `Template` function with the retrieved `CCTS` instance to generate a BPEL template with default elements;

Line 4: we call the follow-up `Traversal` function to fill the above template with detailed elements via CCTS traversal;

Next, we will elaborate on `Template` and `Traversal` functions, respectively, to detail the BPEL generation process with a mapping example in Fig. 6.7.
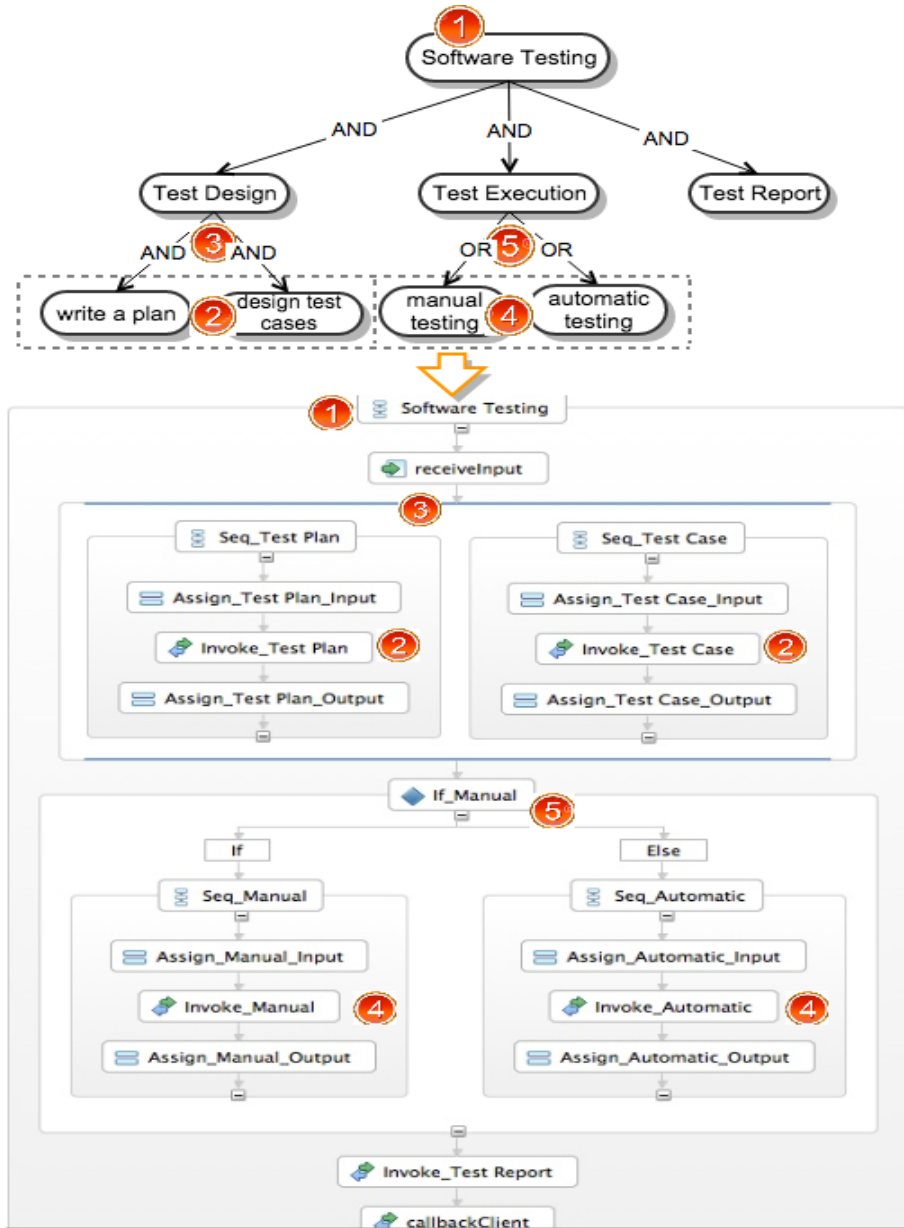
Figure 6.7: CCTS-to-BPEL Transformation Process Example

Algorithm 1: CCTS-to-BPEL Transformation

---

1 **Function** `Transform`(*Plan plan*, *Artifact bpel*)

2   *CCTS root = getRootCcts(plan);*

3   *Template(root, bpel);*

4   *Traversal(root, bpel);*

5 **Function** `Template`(*CCTS root*, *Artifact bpel*)

6   *Artifact bpelWS = wsdlGenerator(root);*

7   *defaultBpelElementsGenerator(source, bpelWS, bpel);*

8 **Function** `Traversal`(*CCTS ccts*, *Artifact bpel*)

9   **if** *isAtomic(ccts)* **then**

10   *invokeReceiveGenerator(ccts);*

11   **else**

12   *List < CCTS > children = getSubCcts(ccts);*

13   **for each** *CCTS subCcts* **in** *children* **do**

14   *Traversal(subCcts, bpel);*

15   **end for**

16   *Operator operator = getDecompositionOperator(ccts);*

17   **switch** *operator* **do**

18   **case** *OR* **do**

19   *ifElseGenerator(operator, bpel);*

20   *break;*

21   **end case**

22   **case** *AND* **do**

23   **if** *anyDataFlowDependencyIn(children)* **then**

24   *sequenceGenerator(children, bpel);*

25   **else**

26   *flowGenerator(children, bpel);*

27   **end if**

28   *break;*

29   **end case**

30   **case** *default* **do**

31   *break;*

32   **end case**

33   **end switch**

34   **end if**

Template Function: This function aims to generate a BPEL specification template with some default elements. Within this function,

Line 6: given the passed root CCTS, we generate a WSDL artifact as a service interface between the target BPEL process and the requester; therefore, (s)he can kick off the execution phase via a service request later on. This WSDL generation process is similar to the `Activity-to-WSDL` mapping mechanism explained in Sec. 5.1.3.

Specifically:

abstract WSDL elements generation: WSDL `type` is generated based on our human artifact data model, while the pair-wise service `request message` and `response message` are generated based on the `CCTS workload` - i.e. the pair-wise input and output, respectively. Besides, each service `operation` and its `portType` wrapper are generated based on `CCTS goal_tag`.

concrete WSDL elements generation: WSDL `binding` is auto-generated based on WS-Addressing/SOAP protocol, while WSDL `service` is generated accordingly with a default endpoint.

Line 7: we generate some default BPEL elements, e.g. the main `<bpel:sequence>` in step one of Fig. 6.7.

With a template generated from above, we will fill it with some concrete BPEL service and control-flow related elements through the following `Traversal` function.

Traversal Function: This function aims to map our CCTS data model entities, e.g. `AND`, `OR`, and `CCTS`, into those service and control-flow related elements in BPEL. To this end, we follow the classic `post-order` tree traversal - i.e. children nodes first;

then the parent node, to recursively traverse the given fully-structured CCTS tree for BPEL elements generation. Specifically, within its function body,

Line 9 - 15: during the CCTS tree traversal, we recursively retrieve a list of atomic sub-CCTS and map each of them into a pair of `<bpel:invoke>` and `<bpel:receive>` elements that will send service request to, and receive service response from a CWS bound with the current atomic sub-CCTS. The mapping process can be illustrated by step two and four of Fig. 6.7. For instance, the key attributes of `<bpel:invoke>` and `<bpel:receive>`, e.g. `partnerLink`, `inputVariable`, and `outputVariable`, are generated based on the bound CWS interface - i.e. WSDL.

Line 16 - 17: when we finish the traversal of all atomic sub-CCTS and are back to their parent, we retrieve the decomposition operator that is used to divide the current CCTS into those atomic ones. Then we map this operator into the control-flow activities in BPEL accordingly. It is worth noting that we do not generate a comprehensive list of BPEL control-flow activities. Instead, we only target at `<bpel:sequence>`, `<bpel:if>`, and `<bpel:flow>` activities, given the simplicity design of our decomposition operator and the data-flow dependencies that it brings into CCTS.

Line 18 - 29: we generate `<bpel:if>` element when it is `OR` operator. As to `AND` operator, we generate `<bpel:sequence>` or `<bpel:flow>`, depending on if any data-flow dependency exists among sub-CCTS, e.g. we generate `<bpel:sequence>` if any output of a sub-CCTS is the input of another. The mapping process can be illustrated by step three and five of Fig. 6.7.

By designing a separate plan phase consisting of both human-level planning and machine-level planning stages, we can have a clearly-defined task model that details the requester's expectation and facilitates the execution management later on. Further, through auto-transforming this high-level human task model into a low-level

service orchestration schema, we can help the requester auto-coordinate multiple crowd workers during the later execution phase.

6.3 Execution Phase – CWS Coordination and Interaction

During this phase, our *SOC4Crowd* framework auto-coordinates the crowd and requester for the originally advertised task based on its planning result from the above. To this end, we design the following *Coordination Model* and its underlying *Interaction Protocol*.

6.3.1 Coordination Model

As explained earlier, the *coordination* feature introduced into our *SOC4Crowd* framework is reflected via the interaction management. From a global perspective, this interaction management is realised through orchestrating multiple crowd workers and managing their work dependencies in a BPEL runtime of our system. From the individual perspective, this interaction management is realised through handling the service request and response between the requester and our *SOC4Crowd* framework, as well as between each crowd worker and *SOC4Crowd*, respectively. Next, we will elaborate on each of those individual interactions through *sequence design*.

Interactions between Requester and SOC4Crowd

As recalled from Sec. 6.1.2, this interaction is started when the requester manually kicks off the execution phase through sending an initial service request to our

SOC4Crowd framework. Then (s)he will get our system response each time when a crowd worker finishes his/her human service. The detailed sequence design of this interaction can be seen in Fig. 6.8.
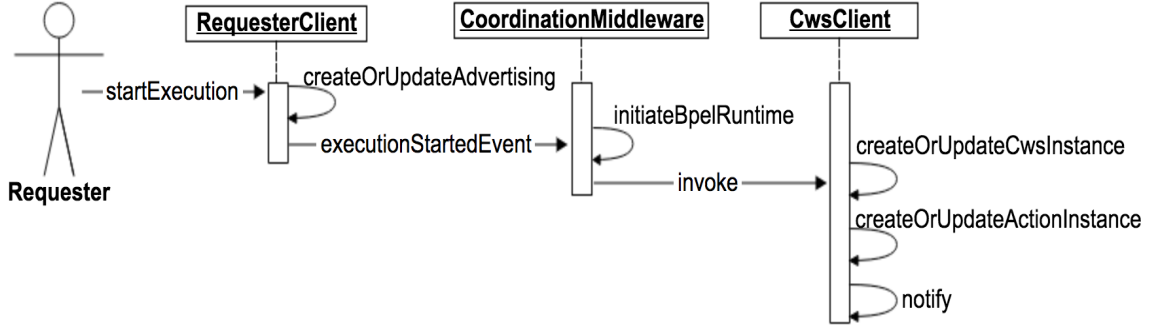

Figure 6.8: Interactions from the requester to SOC4Crowd

As illustrated, the sequence is started when the public `startExecution` service of `RequesterClient` application is triggered by the requester. Internally, this client application calls its private `createOrUpdateAdvertising` operation to update the `status` attribute of the original `Advertising` entity with `executing` value. As such, each copy of this Advertising entity - i.e. `root CCTS`, is removed from the dashboard of those crowd workers who got involved in the previous plan phase, since no planning actions are allowed at this stage.

`RequesterClient` then fires a `EXECUTION_STARTED` event to `CoordinationMiddleware` who subsequently deploys and starts a BPEL process whose specification was generated from the previous plan phase. `CoordinationMiddleware` calls its private `initiateBpelRuntime` to do so, e.g. calling the related APIs of *Apache ODE*. Since then, the deployed BPEL process is being executed at runtime to `invoke` the bound CWS.

When `CwsClient` receives a service request from `CoordinationMiddleware`, it calls its internal `createOrUpdateCwsInstance` operation to initialize a `CwsInstance` entity to record the request details, e.g. request content and its requester information,

for the later response use. Also, it creates an instance of human `Action` entity through its private `createOrUpdateActionInstance` operation. In doing so, the crowd worker can see the new created action on his/her dashboard and respond it with the produced content. In the meantime, `CwsClient` also `notify` the crowd worker about the coming request through both a dashboard message and email.

Interactions between Crowd Worker and SOC4Crowd

This interaction occurs when the crowd worker finishes his/her human work and responds the earlier service request from our system. We design its detailed sequence in Fig. 6.9
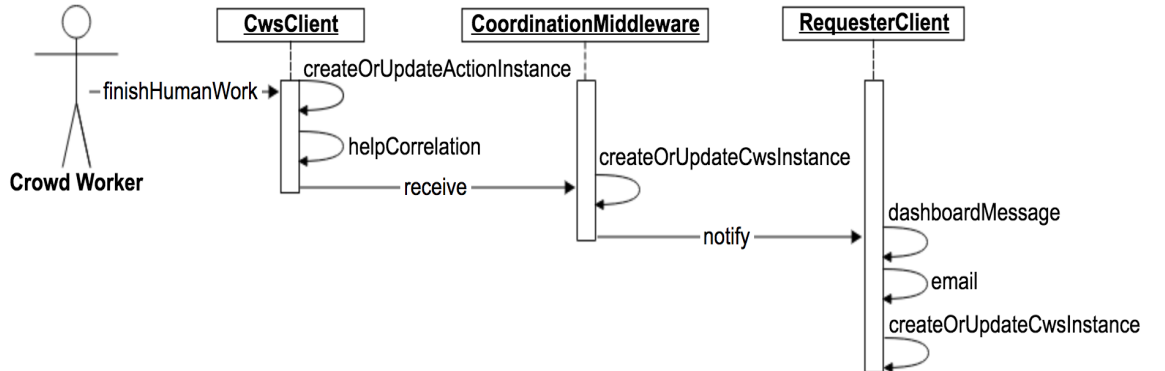
Figure 6.9: Interactions from the crowd worker to SOC4Crowd

As illustrated, the sequence is started when a crowd worker triggers the public `finishHumanWork` service of `CwsClient` application. Internally, this client application calls its private `createOrUpdateActionInstance` operation to instantiate a `HumanArtifact` that encapsulates the produced content as the `output` of current human `ActionInstance`. Before it sends a response with this `output`, it needs to help `CoordinationMiddleware` correlate this response with its earlier request, given the asynchronous service interaction; otherwise `CoordinationMiddleware` won't route the message properly during the BPEL process execution.

To do so, within its private `helpCorrelation` operation, `CwsClient` retrieves the request context, e.g. requester information, the bound CCTS, etc., from the current `CwsInstance`; then it constructs the response message accordingly. Then it calls back `CoordinationMiddleware` with that message through the provided callback service and operation referenced in `receive` BPEL activity.

When our `CoordinationMiddleware` receives the callback response, it firstly calls its `createOrUpdateCwsInstance` operation to record the response details in the current `CwsInstance` entity. Then it asks `RequesterClient` to `notify` the requester about an update on the advertised task execution. After that, the requester can view the produced content and evaluate the finished CWS service by `rating` it through `createOrUpdateCwsInstance` operation.

As we can see from the above sequence designs, human `notification` and message `correlation` play a key role in realising those interactions. To support those two key features, we design an `Interaction Protocol` in the following section.

6.3.2 Interaction Protocol

As explained earlier, during the execution phase, the interaction occurs at both human and machine levels. For the former, it occurs between the human actor (i.e. the requester and crowd) and our *SOC4Crowd* system. For the later, it occurs between system modules - i.e. `RequesterClient`, `CoordinationMiddleware`, and `CwsClient` applications. To support both, we provide `notification` and `correlation` features, respectively. To realise those features, here we design a protocol consisting of a message schema and its processing specification.

Message Schema

Since we are applying the mature *WS-** technologies, e.g. *WSDL* and *BPEL*, into crowdsourcing domain, the message schema has been designed as *SOAP* envelope.

```
:soapenv:Envelope
    xmlns:soapenv="http://schemas.xmlsoap.org/soap/envelope/"
    xmlns:sof="http://interface.cws.soc4crowd/softwareTesting"
    xmlns:cws="http://soc4crowd/cws">
    <soapenv:Header
        xmlns:wsa="http://www.w3.org/2005/08/addressing">
        <cws:provider>
            <cws:providerName>shawn</cws:providerName>
            <cws:providerContact>shawn.xiao.lu@gmail.com</cws:providerContact>
        </cws:provider>
        <wsa:Action>http://interface.cws.soc4crowd/softwareTesting/manualTesting</wsa:Action>
        <wsa:ReplyTo>
            <wsa:Address>http://localhost:8088/mockManualTestingCallbackBinding</wsa:Address>
        </wsa:ReplyTo>
        <wsa:MessageID>uuid:8f2d82f5-4692-40d3-abd4-6d7e9d270e87</wsa:MessageID>
        <wsa:To>http://interface.cws.soc4crowd/softwareTesting/manualTesting</wsa:To>
    </soapenv:Header>
    <soapenv:Body>
        <sof:manualTestingInput>
            <cws:job>
                <cws:jobId>1001</cws:jobId>
                <cws:jobName>Manual Testing</cws:jobName>
                <cws:jobDescription>manual testing for a website</cws:jobDescription>
            </cws:job>
            <!--1 or more repetitions:-->
            <cws:artifact>
                <cws:artifactDescription>Test Plan</cws:artifactDescription>
                <cws:artifactLocation>http://JIRA ...</cws:artifactLocation>
            </cws:artifact>
        </sof:manualTestingInput>
    </soapenv:Body>
:/soapenv:Envelope>
```

Figure 6.10: A message example as per protocol

Specifically,

Header: we utilize those `WS-Addressing` elements, e.g. `MessageID`, `ReplyTo`, etc., in `SOAP Header` for the machine-level callback. Further, to support the human-level notification, we extend WS-Addressing by adding some of our *SOC4Crowd* elements, e.g. `CwsProvider`.

Body: in order to minimize the payload size of our SOAP message for a sound system performance in the distributed computing environment, we do not encapsulate the binary content of `HumanArtifact` into the SOAP body. Instead,

we put its web location - i.e. URL, for the later content retrieval by our human actors themselves. Also, we include the task identity in the body for correlation.

An example of the above customized SOAP message can be seen in Fig. 6.10.

Message Processing

Here we specify an interaction procedure that instructs how we process the above message for both *notification* and *correlation* support.

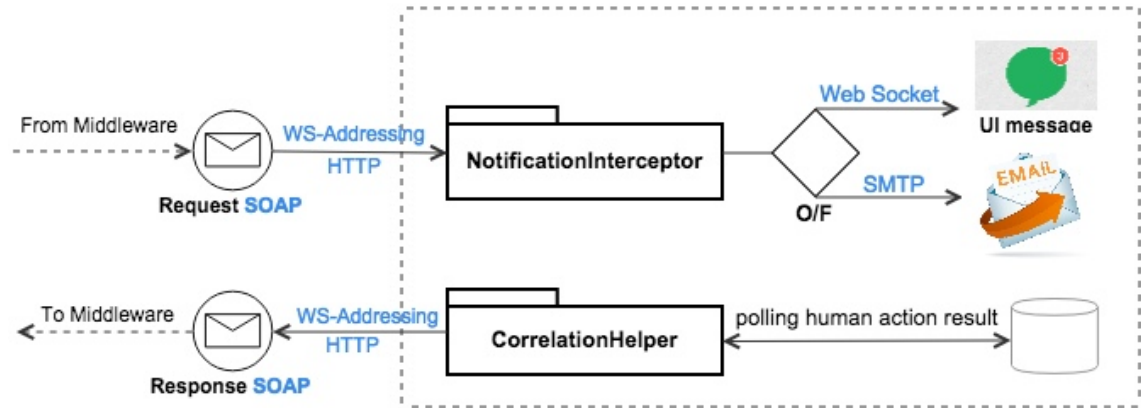Figure 6.11: Message processing as per protocol

This procedure is designed in Fig. 6.11. As illustrated, it specifies the following.

Notification Support. A `NotificationInterceptor` component needs to be realised in both `CwsClient` and `RequesterClient` applications to handle requests from `CoordinationMiddleware` through notifying their end users - i.e. the crowd worker and requester.

Specifically,

UI Message Notification: when the human actor is logged into our system, our `NotificationInterceptor` component parses the incoming message header to retrieve the user name and identity information. Given that, it constructs a JSON message, which is front end friendly, and pushes it onto the dashboard user interface over `Web Socket` protocol.

Email Notification: `NotificationInterceptor` component constructs an email and sends it to the human actor over `SMTP` protocol when the actor is offline. The address given in the message header is used.

As notified, our human actor will proceed his/her action to push the execution phase forward, e.g. the crowd worker will perform the human work and respond the service request.

Correlation Support. Due to the time-consuming nature of human service, we cannot design the interaction as a long-lasting, synchronous one; instead, it has to be asynchronous. That means, after our `CoordinationMiddleware` sends a request, it should not get blocked by the late response and it should be able to correlate the late response with its earlier request to route messages correctly.

To assist `CoordinationMiddleware` with its correlation, a `CorrelationHelper` component is realised in `CwsClient` to construct a proper response message with the required correlation information. Specifically, it keeps polling the human action result from the content repository to get a human work completed signal. Once it gets that signal, it retrieves the CCTS or job ID from the earlier request recorded in `CwsInstance` entity and put it into the response SOAP body. Then it can be later used as the correlation identifier by our `CoordinationMiddleware` while receiving the callback response.

With this underlying `Interaction Protocol`, our `Coordination Model` can be re-

alised throughout the execution phase.

So far, we have specified our operation model - i.e. `Two-Phase Crowdsourcing`
`Management`. Particularly, in each phase specification, we have detailed how each
module of the system architecture interacts with others to manipulate the CCTS and
CWS data model through sequence designs. In the next chapter, we will elaborate
on its implementation.

Chapter 7

Implementation and Use Case

In this chapter, to demonstrate how the design can be realised using existing technologies, we present the details of our prototypical implementation.We also illustrate how the system works by a walk-through of a user case scenario.

7.1 Architecture and Technologies

In this section, we firstly explain our *SOC4Crowd* system architecture by following the design presented in the previous chapters. Then we introduce a stack of technologies used for implementation.

7.1.1 System Architecture

As introduced in Sec. 6.1, the *SOC4Crowd* framework consists of three distributed web-based applications: *CwsClient*, *CoordinationMiddleware*, and *RequesterClient*. These components use one centralized data repository. The anatomy of its architec-
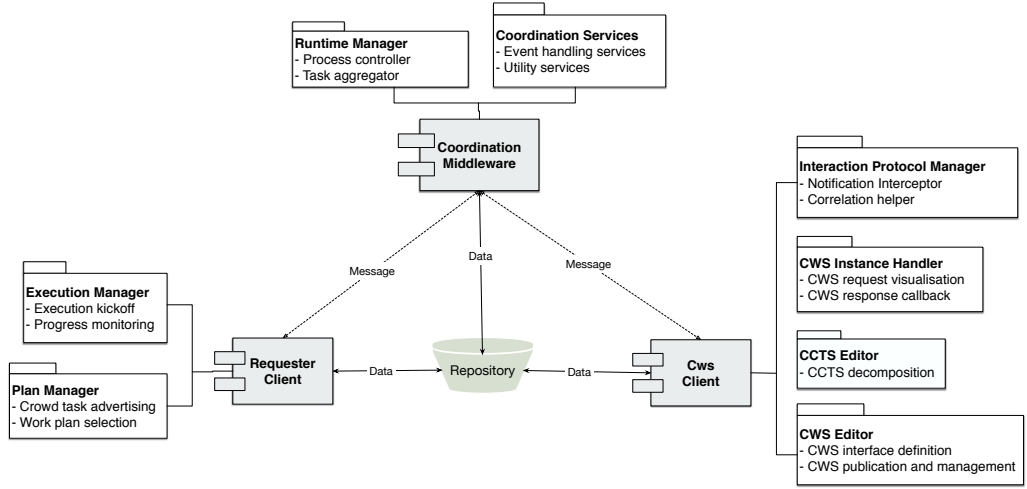
Figure 7.1: SOC4Crowd architecture with key modules

ture can be seen in Fig. 7.1. Next, we will briefly explain the role of each module in this architecture.

CwsClient Application

CwsClient is an application designed for the crowd workers. It provides the crowd workers with custom designed user interfaces to manipulate the CWS and CCTS data models, and to participate in the two-phase crowdsourcing lifecycle. To support diverse devices and environments on the client side, it is implemented as a web-based application consisting of the following modules:

CWS Editor: `CWS Editor` allows the crowd workers to define and expose their human services to the public. Using the editor, the crowd workers declare actions that they are capable of performing. Behind the scene, the declarations are 'mapped' and translated automatically to an appropriate WSDL artifact. This part of implementation follows the *Activity-to-WSDL* mapping design in Sec. 5.1.3. As a result, entities in our CWS data model, such as `Activity`, `Action`, and `CwsInterface`, are instantiated.

Other elements of the editor support richer descriptions of the human service such as provider profile uploads, specifying the service availability and cost, etc. Importantly, the editor also provides a service publication function whereby the declared human service can be registered with the *Coordination-Middleware* application for discovery. As a result, entities in our CWS data model, such as `CwsContext` and `Provider`, are instantiated. Through `CWS Editor`, crowd workers can maintain and update their human services as their capabilities evolve.

CCTS Editor: `CCTS Editor` is used, during the planning phase of crowdsourcing lifecycle, to decompose the complex work and ultimately create the CCTS data model. The user interface provides a simple editor to use *AND* or *OR* operators to create and coordinate sub tasks. The details of each task – description, deadline, input and output – are defined here as well. In the back end, it communicates with *CoordinationMiddleware* to notify *RequesterClient* about a new work plan generated.

Through the `CCTS Editor`, each crowd worker can contribute their own task plan to the requester in the planning phase. As a result, multiple CCTS instances are generated and one of them would be later chosen as the final work plan to guide the execution phase.

CWS Instance Handler: This module is responsible for CWS request and response handling during the execution phase of the lifecycle. Through this module, each crowd worker can have their own ''workbench'' to handle their human service request, e.g. receiving the service input and responding with an output, during the execution phase.

Interaction Protocol Manager: This module manages notifications and message correlations during the execution phase of the lifecycle. This module implements the design of the interaction protocol between *CwsClient* and *CoordinationMiddleware* applications, which is presented in Sec. 6.3.2.

As we can see from all the above modules, *CwsClient* application acts as an interface between crowd workers and our *SOC4Crowd* framework. It empowers crowd workers to perform many crowdsourcing activities to operate both CWS and CCTS data models through the whole two-phase crowdsourcing lifecycle, e.g. task decomposition in the plan phase and service interaction in the execution phase. Next we will introduce the anatomy of our *CoordinationMiddleware* to briefly show how it is implemented to coordinate and interact with those client applications in our architecture.

Coordination Middleware

Given its role as a coordinator between *CwsClient* and *RequesterClient*, the *CoordinationMiddleware* application is implemented as a server application providing many runtime services to its two client applications.

Coordination Services: to manage the two-phase crowdsourcing lifecycle as a whole, *CoordinationMiddleware* implements an event handling mechanism to represent and manage any state change during the whole crowdsourcing process. It exposes a uniform RESTful service to the two client applications to communicate the events, such as CWS and CCTS data updates, or work progress updates. For instance, when a crowd task has been advertised through *RequesterClient* application, a `TASK_ADVERTISED` event is triggered and that uniform RESTful service gets called to update CCTS data model and broadcast the new advertising to the crowd.

Runtime Manager: this module implements two important functions during the execution phase of the lifecycle. It automatically deploys, start, suspend, and complete a BPEL process, according to the state change of a crowdsourcing process. For instance, when the requester kicks off the execution phase of a

crowdsourcing process, it automatically deploys and starts a BPEL process whose definition is generated from the plan phase. It also provides **Task Aggregator** which aggregates the crowd task results when any of them is generated as the process executes.

RequesterClient Application

This is a web application serving as an interface between the human requester and our *SOC4Crowd* framework. It aims to support the requester to participate in the planning and execution phases.

Plan Phase Manager This module aims to help the requester initiate and participate in the plan phase of a crowdsourcing process. The requester can advertise a complex crowd task with descriptions, deadline, the input and expected output, etc. The plan manager then initiate the CCTS data model, e.g. `Advertising` entity, and asks *CoordinationMiddleware* to broadcast this new advertising to the crowd. Towards the end of the planning phase, it also provides a feature that allows the requester to review each candidate plan and make the final selection. The plan manager will update the CCTS data model and asks *CoordinationMiddleware* to generate a corresponding BPEL artifact.

Execution Phase Manager This module lets the requester initiate and monitor the execution phase of a crowdsourcing process. To this end, a user interface is provided for the requester to kick off the execution with the initial input. Correspondingly, the back-end program constructs a SOAP message and sends it to *CoordinationMiddleware* to initiate a BPEL process at runtime.

7.1.2   Implementation Technologies

Here, we introduce the technologies used to realise the above system architecture.

Application	Technologies used
Repository	*PostgreSQL*
CwsClient	*HTML, CSS, Javascript, AJAX, Java Web, WebSocket, Atmosphere, Jersey, Apache CXF*
RequesterClient	*HTML, CSS, Javascript, AJAX, Java Web, Jersey, Apache CXF*
Middleware	*Apache ODE, Jersey, Apache CXF*

Figure 7.2: A stack of key technologies used for implementation

As shown in Fig. 7.2, first of all, we use *PostgreSQL*[1] as the central data repository implementation, since we store a mix of schematic content, e.g. *CCTS* and *CWS*, and schema-less documents, e.g. JSON messages and user-generated artifact.

Both *RequesterClient* and *CwsClient* are realised as web applications using a Java-based, RESTful service application development framework, and Javascript/CSS for front-end development.

In terms of our *CoordinationMiddleware*, given the designed execution phase wherein a BPEL process is deployed and executed, we implement it as an extension to a mature BPEL runtime. We chose *Apache ODE*[2] because it is an open-source and lightweight solution with a well-formed developer community; also it is more suitable for the development of this proof of concept in our research, when compared with other complex commercial products. All SOAP and BPEL based interactions are implemented using *Apache CXF*[3] framework.

---

[1]https://www.postgresql.org/

[2]http://ode.apache.org

[3]http://cxf.apache.org/

We have chosen a set of open-source frameworks to realise the communications within and between applications. For instance, within either *CwsClient* or *Requester-Client*, the communication between their front-end and back-end is realised via both *PULL* and *PUSH* strategies. As to *PULL* strategy, we use AJAX (i.e. asynchronous JavaScript and XML) call at the front end to pull data from the back end to implement features like displaying a list of crowd tasks or services on the dashboard. As to *PUSH* strategy, we choose *Atmosphere*[4] framework that supports *WebSocket* protocol at both front and back ends to implement features like push notification.

Next, we will demonstrate how to apply those technologies as above to implement those features and modules of our system architecture in Sec. 7.1.1.

7.2 Implementation Details

In this section, we introduce some programming code fragments to explain how we implement those key features and modules of each application in our system architecture.

7.2.1 CwsClient Web Application

As its name suggests, *CwsClient* acts as an interface between crowd workers and our *SOC4Crowd* framework. Through this interface, crowd workers can participate in the crowdsourcing process and perform many activities, e.g. decomposing CCTS in plan phase and handling CWS request in execution phase.

---

[4]https://github.com/Atmosphere/atmosphere

CWS Editor – CWS Definition via XForm and XSLT

Besides the conventional web application development effort, we have used *XForm*[5] and *XSLT*[6] to ease the transformation of the CWS definition data collected from the UI to the data model. *XForm* is a markup language that can be used to enhance a web form to not only collect the data but also process it at the same time given a pre-defined data model, while *XSLT* is a transforming language that is often used to convert one XML document into another. Essentially, this helps `Activity-to-WSDL` mapping mechanism presented in Sec. 5.1.3 in that when a crowd worker uses the UI to declare his human activity details (e.g., *software_test* activity and its *manual_test* action), the details are automatically bound to *XForm*, and the resulting XML document can be directed to an XSLT program that maps the human actions to WSDL specification.


```
<!-- Types -->
<xsl:element name="types">
    <xsl:element name="xsd:schema">
        <xsl:attribute name="targetNamespace">
            http://interface.cws.soc4crowd/<xsl:value-of select="concat(
            translate(substring(Activity/Tag,1,1), $uppercase, $lowercase), substring(Activity/Tag,2))"/>
        </xsl:attribute>
        <xsl:element name="xsd:import">
            <xsl:attribute name="namespace">http://soc4crowd/cws</xsl:attribute>
            <xsl:attribute name="schemaLocation">cws.xsd</xsl:attribute>
        </xsl:element>
        <xsl:for-each select="*/Input">
            <xsl:element name="xsd:element">
                <xsl:attribute name="name">
                    input_<xsl:value-of select="concat(
                    translate(substring(Tag,1,1), $uppercase, $lowercase), substring(Tag,2))"/>
                </xsl:attribute>
                <xsl:attribute name="type">cws:<xsl:value-of select="Type"/></xsl:attribute>
            </xsl:element>
        </xsl:for-each>
        <xsl:for-each select="*/Output">
            <xsl:element name="xsd:element">
                <xsl:attribute name="name">
                    output_<xsl:value-of select="concat(
                    translate(substring(Tag,1,1), $uppercase, $lowercase), substring(Tag,2))"/>
                </xsl:attribute>
                <xsl:attribute name="type">cws:<xsl:value-of select="Type"/></xsl:attribute>
            </xsl:element>
        </xsl:for-each>
    </xsl:element>
</xsl:element>
```

Figure 7.3: Activity-to-WSDL mapping rule fragment

A fragment of the XSLT code is shown in Fig. 7.3. It shows that we generate

---

[5]www.w3.org/community/xformsusers/wiki/XForms_2.0

[6]www.w3.org/TR/xslt

each WSDL "type" element corresponding to each human action's input and output. These will be referenced later by WSDL "message" elements that map human actions.

```xml
<definitions name="SoftwareTesting" targetNamespace="http://interface.cws.soc4crowd/software_test"
    xmlns="http://schemas.xmlsoap.org/wsdl/" xmlns:xsd="http://www.w3.org/2001/XMLSchema"
    xmlns:wsdl="http://schemas.xmlsoap.org/wsdl/" xmlns:soap="http://schemas.xmlsoap.org/wsdl/soap/"
    xmlns:wsaw="http://www.w3.org/2006/05/addressing/wsdl"
    xmlns:cws="http://soc4crowd/cws"
    xmlns:st="http://interface.cws.soc4crowd/software_test">

    <types>
        <xsd:schema targetNamespace="http://interface.cws.soc4crowd/software_test">
            <xsd:import namespace="http://soc4crowd/cws" schemaLocation="cws.xsd" />
            <xsd:element name="input_test_case" type="cws:artifact" />
            <xsd:element name="output_test_report" type="cws:artifact" />
        </xsd:schema>
    </types>
```

Figure 7.4: WSDL types fragment generated by XSLT

Further, Fig. 7.5 shows a screen capture of the results of an online WSDL validator[7]. The result illustrates that the WSDL artifact generated by `CWS Editor` is syntactically correct and complete.

```
***Activity Declaration is saved successfully
***Action Declaration is saved successfully
***Begin to generate WSDL file
***WSDL file generated successfully          WSDL generated
***Validate WSDL
Retrieving document at '/Users/yifanfan/Documents/workspace/.metadata/.plugins/
org.eclipse.wst.server.core/tmp0/wtpwebapps/cws/WEB-INF/classes/file/target.wsdl',
relative to '/Users/yifanfan/Documents/workspace/.metadata/.plugins/
org.eclipse.wst.server.core/tmp0/wtpwebapps/cws/WEB-INF/classes/file/cws.xsd'.
Retrieving schema at 'cws.xsd', relative to 'file:/Users/yifanfan/Documents/
workspace/.metadata/.plugins/org.eclipse.wst.server.core/tmp0/wtpwebapps/cws/WEB-
INF/classes/file/target.wsdl'.                  Validation of
+++:Passed Validation : Valid WSDL:+++        WSDL file
***The WSDL file is valid
***content in System_Artifact is not null
***System_Artifact is saved successfully
***Cws_Interface is saved successfully
***New Cws is saved successfully
```

Figure 7.5: WSDL artifact validation

---

[7]https://wiki.eclipse.org/Using_the_WSDL_Validator_Outside_of_Eclipse

CWS Editor – CWS Publication

Listing 7.1 shows the RESTful service code that publishes a completed CWS from `CWS Editor`.

Listing 7.1: Publishing CWS

```java
@Path("/cws/{id}/publication")
@POST
@Consumes(APPLICATION_JSON)
public Response publish(@PathParam("id") long id, CwsContextDTO cxtInfo) {
    try {
        // validate the input
        checkArgument(id > 0, "CWS ID must be greater than 0!");
        checkIfCompleteContext(cxtInfo);
        // persist cws context and register it in Middleware
        CwsContext cxtEntity = instantiateCxtEntityBy(cxtInfo);
        long cxtEntityId = repository.save(cxtEntity);
        middleware.registerCws(cxtEntityId, cxtInfo);
        // redirect crowd workers to their dashboard
        return Response.seeOther(new URI("/cws/dashboard")).build();
    } catch (Exception e) {
        return Response.serverError().build();
    }
}
```

```
POST http://127.0.0.1:8080/cws/101/publication

content-type: application/json
accept-encoding: gzip, deflate

{
  "id": 101,
  "cxtInfo": {
    "availability": {
      "isAvailable": true,
      "expiryDate": "31/12/2016"
    },
    "cost": "$100",
    "providerInfo": {
      "id": 201,
      "name": "shawn",
      "contact": "shawn.xiao.lu@gmail.com",
      "profile": ""
    },
    "activityId": 301,
    "tags": [
      "software_test",
      "manual_test"
    ],
    "wsdlLocation": "http://127.0.0.1:8080/cws/software_test?wsdl"
  }
}
```

Figure 7.6: CWS Publication Service Request

7. Implementation and Use Case

After verifying all input, it saves the given CWS context information into the central data repository; then registers it to *CoordinationMiddleware* for service discovery. Fig. 7.6 shows an example of a request that the above code sends to *Coordination-Middleware* for CWS publication.

CCTS Editor – CCTS Decomposition

To enable CCTS decomposition, we implement a user interface to manipulate a CCTS instance. Fig. 7.7 shows an example of a root CCTS instance prior to any decomposition on that UI. As demonstrated, this root CCTS instance has a flat structure without any sub-CCTS.

```json
{
  "name": "RHS system testing",
  "goal_tags": ["software_test", "quality_assurance"],
  "input_tags": ["software_requirement", "system_uat_url"],
  "output_tags":["test_plan_doc", "test_case_doc", "test_report_doc"],
  "deadline": "31/12/2016",
  "budget": "$1000",
  "type": "root_ccts"
}
```

Figure 7.7: A Sample of Root CCTS Instance in JSON

When it is decomposed, the CCTS instance is updated to have a hierarchical structure containing the decomposing operators and sub-CCTS details. Fig. 7.8 shows an example of the decomposed CCTS instance. In the later Sec. 7.3, we illustrate how the front-end UI enables the decomposition to generate these JSON data.

Listing 7.2: CCTS Validation

```java
private CCTS validateAndTransform(CctsJson cctsJson) {
    // ccts value and structure validation
    requireNonNull(cctsJson, "Input CCTS must exist!");
    checkArgument(ifAnyInvalidSubCCTS(cctsJson), "No CCTS can be both parent and child!");
    checkArgument(ifAnyInvalidDeadline(cctsJson), "Any sub-CCTS deadline cannot be later than that
        of parent!");
    checkArgument(ifAnyInvalidBudget(cctsJson), "Budget sum of sub-CCTS cannot be greater than that
        of parent!");
    checkArgument(ifAnyInvalidDataFlow(cctsJson), "Data-flow loop among sub-CCTS is detected!");
    // data format transform
    return CCTS.builder().buildFromJson(cctsJson);
```

```json
{
  "name": "RHS system testing",
  "goal_tags": ["software_test", "quality_assurance"],
  "input_tags": ["software_requirement", "system_uat_url"],
  "output_tags":["test_plan_doc", "test_case_doc", "test_report_doc"],
  "deadline": "31/12/2016",
  "budget": "$1000",
  "type": "root_ccts",
  "decomposition": {
    "operator": "AND",
    "sub-CCTS": [
      {
        "name": "RHS system testing plan",
        "goal_tags": ["test_plan"],
        "input_tags": ["software_requirement"],
        "output_tags":["test_plan_doc"],
        "deadline": "10/12/2016",
        "budget": "$100",
        "type": "sub_ccts"
      },
      ...
    ]
  }
}
```

Figure 7.8: Decomposed CCTS Instance in JSON

```
}
```

Once crowd workers finish their task decomposition and submit it as their plan through the provided UI, a fully structured CCTS instance in JSON format is posted to the back end for validation and persistence. Listing 7.2 shows the overall process of validating the decomposed CCTS instance passed from the front end and transforming it to our back-end data structure for persistence.

CWS Instance Handler and Interaction Protocol Manager

From implementation point of view, the main function of `CWS Instance Handler` in the framework is to validate and transform the task instance data (input, output) from its UI data model (JSON) to the matching human activity/action data model. As we have presented similar concepts previously, we do not repeat the implementation details of the module. In this section, we present how the `Interaction`

`Protocol Manager` is implemented to support the actual SOAP communications between the crowd worker and the coordination middleware.

NotificationInterceptor for human-level notification implementation: This module supports the communications between the instance handler and the coordination middleware by (i) transforming the incoming SOAP message from *CoordinationMiddleware* to a human readable service request and notification (ii) constructing the outgoing SOAP response and callback *CoordinationMiddleware* for the asynchronous message correlation to communication the output of the crowd worker's action. To implement that, we use `Apache CXF Interceptor` related APIs to intercept each incoming SOAP request from *CoordinationMiddleware* and processes it before being visualized in the above `CWS Instance Handler`. Listing 7.3 shows an overview of the implementation.

Listing 7.3: SOAP handling in NotificationIncerceptor

```
@Override
public boolean handleMessage(SOAPMessageContext context) {
    // only handle the incoming request message
    if (!isOutboundMessage(context)) {
        // unmarshalling the incoming SOAP message
        CwsInstanceDTO cwsInfo = unmarshalling(context.getMessage(), CwsInstanceDTO.class);
        // cws data model update
        long activityId = repository.save(buildActivityIntanceBy(cwsInfo));
        long actionId = repository.save(buildActionIntanceBy(cwsInfo));
        repository.save(buildCwsIntance(cwsInfo, activityId, actionId));
        // notification via both WebSocket pushed message and email
        Broadcaster broadcaster = getBroadcasterFactory().lookup(cwsInfo.getProviderContact());
        broadcaster.broadcast(cwsInfo.getAction());
        String mailContent = format(template, cwsInfo.getProviderName(), cwsInfo.getAction());
        mailer.sendSimpleMail(cwsInfo.getProviderContact(), mailContent);
    }
    return true;
}
```

To illuminate the above code fragment, we take an exemplified SOAP request message, as shown in Fig. 7.9, as the input to go through its implementation in the following order.

```xml
<?xml version="1.0" encoding="UTF-8" ?>
<soap:Envelope xmlns:soap="http://www.w3.org/2001/12/soap-envelope"
               soap:encodingStyle="http://www.w3.org/2001/12/soap-encoding"
               xmlns:wsa="http://schemas.xmlsoap.org/ws/2004/08/addressing"
               xmlns:xsd="http://www.w3.org/2000/10/XMLSchema"
               xmlns:soc4crowd="http://soc4crowd.org"
               xmlns:st="http://interface.cws.soc4crowd/software_testing">
    <soap:Header>
        <wsa:To>http://localhost:8888/software_test/manual_test_binding</wsa:To>
        <wsa:Action>http://localhost:8888/software_test/manual_test</wsa:Action>
        <wsa:MessageID>uuid:8f2d82f5-2342-4234-asdf-asdas2ds3sd</wsa:MessageID>
        <wsa:ReplyTo>
            <wsa:Address>
                http://localhost:8888/software_test/manual_test_callback_binding
            </wsa:Address>
        </wsa:ReplyTo>
        <soc4crowd:cwsContext>
            <soc4crowd:cwsId>101</soc4crowd:cwsId>
            <soc4crowd:providerName>Shawn</soc4crowd:providerName>
            <soc4crowd:providerContact>luxiao@cse.unsw.edu</soc4crowd:providerContact>
        </soc4crowd:cwsContext>
    </soap:Header>
    <soap:Body>
        <st:manual_test_request>
            <st:input_test_case_doc>
                <cws:artifact_tag>test_case_doc</cws:artifact_tag>
                <cws:artifact_location>http://drive.google...</cws:artifact_location>
            </st:input_test_case_doc>
        </st:manual_test_request>
        <soc4crowd:cctsId>1001</soc4crowd:cctsId>
    </soap:Body>
</soap:Envelope>
```

Figure 7.9: SOAP request message sample during the CWS interaction

- **Boundary Checking**: we firstly check if the intercepted SOAP message is an incoming request or outgoing response, since we only handle the former and ignore the latter.

- **Data Unmarshalling**: we then extract CWS related information, e.g. the requested human activity and action, human service provider, etc., from the intercepted SOAP message, e.g. `<wsa:Action>` and `<soc4crowd:cwsContext>` in Fig. 7.9, for the further data persistence and notification usage.

- **CWS Data Model Update**: with the extracted information from above, we update our CWS data model - i.e. instantiating `CwsInstance`, `ActivityInstance`, and `ActionInstance` entities. In doing so, the earlier discussed `CWS Instance Handler` module can fetch them from data repository and visualize them for crowd workers.

- **Notification**: with the extracted provider and action information from above, we firstly render it as a push notification on the dashboard of a particular crowd worker through `Broadcaster` APIs provided by *Atmosphere* framework.

In general, those APIs transform our back-end information into the front-end friendly data format, e.g. JSON, and push it onto the front end through *WebSocket* protocol. Then we put those information, e.g. human action and provider, into a pre-defined email template and send it to the target crowd worker.

CorrelationHelper for machine-level correlation implementation   This module implements asynchronous interactions between *CwsClient* and *CoordinationMiddleware*. The implementation of this heavily relies on the callback() approach and *aspect-oriented programming (AOP)* technology. We implement a proxy that intercepts the `finishHumanWork` method of `CWS Instance Handler` process service callback and message correlation. The following code fragment demonstrate the approach (Listing 7.4).

Listing 7.4: callback() and AOP for asynchronous interactions

```
public Object afterMethodInvocation(MethodInvocation methodInvocation) {
    // callback only when finishing human work properly
    Response response = (Response) methodInvocation.proceed();
    if (response.getStatusInfo() != INTERNAL_SERVER_ERROR) {
        // retrieve the request details - i.e. CwsInstance
        HumanActionDTO actionInfo = (HumanActionDTO) methodInvocation.getArguments()[0];
        CwsInstance cwsInstance = retrieveCwsInstanceBy(actionInfo);
        // prepare SOAP response body payload
        HumanArtifact humanArtifact = actionInfo.getArtifact();
        long cctsId = actionInfo.getCctsId();
        // do the callback
        doCallback(getCallbackBindingPort(cwsInstance), getCallbackPortType(cwsInstance),
            getCallbackOperation(cwsInstance), humanArtifact, cctsId);
    }
    return response;
}
```

The above `afterMethodInvocation` method in our `CorrelationHelper` would be auto-invoked prior to the `finishHumanWork` method invocation in `CWS Instance Handler` and its callback processing logic can be detailed as follows.

when to callback: we only do the callback when crowd workers finish their human

work properly. To determine so, we firstly allow the `finishHumanWork` method to proceed to get its invocation result. Then we continue our callback process only when the human work response is not at `INTERNAL_SERVER_ERROR` status.

what to callback: we callback our *CoordinationMiddleware* with two parts of information. That is, (i). the `Human Artifact` from human action output; (ii). the `CCTS ID` from the previous CWS request. The former information is CWS request processing result, while the latter will be used by our *CoordinationMiddleware* later correlating the response with its early sent request in order to route messages properly. Both two parts are encapsulated into the body of SOAP response message.

how to callback: to asynchronously response our *CoordinationMiddleware*, we need the callback `Binding`, `PortType`, and `Operation`, since we are utilizing web service interaction technology. Those key *WS-** information is defined in CWS interface - i.e. WSDL, and can be retrieved from the current CWS instance. Along with the above payload information - i.e. `Human Artifact` and `CCTS ID`, we can then construct a SOAP message and send it to *CoordinationMiddleware* as a callback response.

```xml
<?xml version="1.0" encoding="UTF-8" ?>
<soap:Envelope xmlns:soap="http://www.w3.org/2001/12/soap-envelope"
               soap:encodingStyle="http://www.w3.org/2001/12/soap-encoding"
               xmlns:wsa="http://schemas.xmlsoap.org/ws/2004/08/addressing"
               xmlns:xsd="http://www.w3.org/2000/10/XMLSchema"
               xmlns:soc4crowd="http://soc4crowd.org"
               xmlns:st="http://interface.cws.soc4crowd/software_testing">
    <soap:Header>
        <wsa:To>http://localhost:8888/software_test/manual_test_callback_binding</wsa:To>
        <wsa:Action>http://localhost:8888/software_test/manual_test_callback</wsa:Action>
        <wsa:RelatesTo>uuid:8f2d82f5-2342-4234-asdf-asdas2ds3sd</wsa:RelatesTo>
    </soap:Header>
    <soap:Body>
        <st:manual_test_callback_response>
            <st:output_test_case_doc>
                <cws:artifact_tag>test_result_doc</cws:artifact_tag>
                <cws:artifact_location>http://drive.google...</cws:artifact_location>
            </st:output_test_case_doc>
        </st:manual_test_callback_response>
        <soc4crowd:cctsId>1001</soc4crowd:cctsId>
    </soap:Body>
</soap:Envelope>
```

Figure 7.10: Callback SOAP message sample during the CWS interaction

To demonstrate the above implementation result, Fig. 7.10 is the callback SOAP message corresponding to its earlier request SOAP message in Fig. 7.9. As we can

see, the `<wsa:To>` SOAP header in Fig. 7.10 is derived from the `<wsa:ReplyTo>` SOAP header in Fig. 7.9. And the `<wsa:RelatesTo>` SOAP header in Fig. 7.10 corresponds to the `<wsa:MessageID>` SOAP header in Fig. 7.9. Additionally, both SOAP messages share the same `<soc4crowd:cctsId>` payload information in their body. As such, those two messages can be correlated to each other from both SOAP header and body perspective.

As we can see from the above `NotificationInterceptor` and `CorrelationHelper` implementation, it separates the machine-level communication concern, e.g. message transformation and correlation, from the above `CWS Instance Handler` and helps it focus on human action input and output. In doing so, it realises the earlier designed interaction protocol between *CwsClient* and *CoordinationMiddleware* applications at both human and machine level, which is the underlying and key support for the execution phase of our crowdsourcing lifecycle.

7.2.2   Coordination Middleware

CoordinationMiddleware acts as a central orchestrator managing the crowdsourcing process as a whole. We implement it as a pure back-end application serving *CwsClient* and *RequesterClient* applications.

Crowdsourcing Lifecycle Management

A critical service interface implemented in this application is `Uniform Event-handling Service`, which is developed to track the state change, push the progress forward, and coordinate both *CwsClient* and *RequesterClient* accordingly during the crowdsourcing lifecycle.

To be a little more specific, our two-phased crowdsourcing lifecycle consists of a set of

important states, e.g. a new complex task being advertised, its final work plan being selected, etc. Therefore, managing the crowdsourcing lifecycle is realised through managing the state change. To do so, our `Uniform Event-handling Service` is provided for either *CwsClient* or *RequesterClient* to trigger the state change with a specific event. Then the service implementation handles the passed event through content persistence and participants coordination, e.g. updating CCTS data model and notifying *CwsClient* about a new crowd task being advertised by *CwsClient*. Next, we will detail the implementation of `Uniform Event-handling Service` as a typical coordination service in our *CoordinationMiddleware*.

Crowdsourcing lifecycle Events   We firstly define a base event data structure whose detailed schema can be seen in the following code fragment (Listing 7.5).

Listing 7.5: Events in SOC4Crowd lifecycle

```java
public abstract class Event<T> {
    private final EventType eventType;
    private final T content;
    protected Event(EventType eventType, T content) {
        this.eventType = eventType;
        this.content = content;
    }
    public final EventType getEventType() {
        return eventType;
    }
    public final T getContent() {
        return content;
    }
    public static enum EventType {
        TASK_ADVERTISED,
        PLAN_SUBMITTED,
        PLAN_SELECTED,
        EXECUTION_STARTED,
        EXECUTION_STOPPED,
        EXECUTION_COMPLETED
    }
}
```

As shown above, the abstract `Event<T>` class is our base event data structure. It consists of two fields - i.e. `eventType` and `content`, which represents the crowd-

sourcing state change and specific changing content, respectively. For instance, the `TASK_ADVERTISED` event type denotes that a new crowd task has been advertised. As you may notice, the event `content` here is marked as a generalized type without any class definition. That is because it needs to be specified by the concrete subclass extending the above abstract `Event<T>`. As an example, the concrete `TASK_ADVERTISED` event class is defined as follows (Listing 7.6).

Listing 7.6: A Concrete Event Class

```java
public final class TaskAdvertisedEvent extends Event<TaskAdvertisingContent> {
    public TaskAdvertisedEvent(TaskAdvertisingContent content) {
        super(EventType.TASK_ADVERTISED, content);
    }
    public static final class TaskAdvertisingContent {
        long advertisedTaskId;
        String advertisedTaskName;
        Date advertisedTaskDeadline;
        String advertisedTaskBudget;
        List<Tag> advertisedTaskTags;
    }
}
```

As we can see, the above `TaskAdvertisedEvent` class extends the base `Event<T>`, and specifies its concrete event `content` as its nested `TaskAdvertisingContent` class with the task advertising details, e.g. the identity and name of new advertised task. These classes are instantiated by either *CwsClient* or *RequesterClient*. Then subsequently, they are delegated to the right event handler to update the new crowdsourcing state, notify the corresponding participants, and drive the crowdsourcing progress forward.

EventHandler: the crowdsourcing progress driver `EventHandler` accepts an event and update the crowdsourcing state accordingly. Therefore it is the crowdsourcing progress driver that pushes the process forward. We firstly implement a base handler that defines the skeleton structure. Then each concrete handler that accepts each concrete event extends that base handler. As an example, a concrete

EventHandler class, `AdvertisingEventHandler`, is shown in Listing 7.7.

Listing 7.7: A concrete EventHandler

```java
public final class TaskAdvertisingEventHandler extends EventHandler<TaskAdvertisedEvent> {
    @Override
    public void process() {
        // event validation
        if (event.getEventType() != TASK_ADVERTISED) {
            throw new IllegalStateException("Can only handle 'TASK_ADVERTISED' event !");
        }
        // CCTS data model update
        TaskAdvertisingContent advertisingContent = event.getContent();
        Advertising entity = instantiateBy(advertisingContent);
        repository.save(entity);
        // find a group of interesting CWS and notify them
        List<CWS> interestedCWS = findInterestedCwsBy(advertisingContent);
        doNotification(interestedCWS);
    }
}
```

As illustrated above, it extends the base `EventHandler` class with the specified `TaskAdvertisedEvent`. Then it implements its own event process method as follows.

Event Validation: to make sure we are handling the right event, here we check if the passed event is `TASK_ADVERTISED` type.

Content Persistence: to update the new crowdsourcing state due to the passed event, we update the data repository with new content from the passed event. In this particular `TASK_ADVERTISED` case, we update our CCTS data model with the new `Advertising` entity.

CWS Notification: to make *CwsClient* be aware of the new crowdsourcing state triggered by *RequesterClient*, we broadcast the new advertising to the crowd and notify the potential crowd workers who may have their interests in planning it later. As such, we coordinate both *CwsClient* and *RequesterClient* participants to be on the same page, regarding the new crowdsourcing state and follow-up activities. In doing so, we actually push the crowdsourcing progress forward through the event handler.

7. Implementation and Use Case

The lifecycle events and their handlers are the internal building blocks of *CoordinationMiddleware* coordinating a crowdsourcing process. Besides, it also exposes a RESTful service as an interface for both *CwsClient* and *RequesterClient* participants to trigger the event and interact with each other.

Runtime Manager: the executing process controller and administrator

For the actual execution of a task, *CoordinationMiddleware* provides a runtime manager to control and administrate the crowdsourcing process in execution, e.g. deploy, start, and stop the process. In doing so, we separate the low-level web service and BPEL process management from the high-level crowdsourcing management for our requester.

We build *CoordinationMiddleware* on top of an open-source BPEL runtime *Apache ODE*. Therefore, we realise our runtime manager through the APIs provided by *Apache ODE*. Next, we take the process deploy and start as an example to detail its implementation.

In order to separate the low-level WS-BPEL process deployment from the high-level crowdsourcing management and help our requester focus on crowd task plan and execution only, our `Runtime Manager` in *CoordinationMiddleware* realises the WS-BPEL process auto-deployment, when the requester kicks off the execution phase. We can see its specific implementation through the following code fragments.

```java
public void deploy(Plan plan) {
    // validate input and retrieve BPEL artifact content
    checkNotNull(plan, "The given plan must exist!");
    checkArgument(plan.isFinal(), "The given plan must be finalized!");
    checkNotNull(plan.getBpel(), "The given plan must be transformed!");
    byte[] content = plan.getBpel().getContent();
    // init process management API (i.e. 'pmapi') of apache ODE
    OMFactory factory = OMAbstractFactory.getOMFactory();
    OMElement root = factory.createOMElement("deploy",
        factory.createOMNamespace("http://www.apache.org/ode/pmapi", "pmapi"));
    OMElement zipPart = factory.createOMElement("package", null);
    OMElement zipElmt = factory.createOMElement("zip", null);
```

```
        // encapsulate BPEL content into zip element to be ready for deploy
        OMText zipContent = factory.createOMText(Base64.encode(content), "application/zip", true);
        zipElmt.addChild(zipContent);
        zipPart.addChild(zipElmt);
        root.addChild(zipPart);
        // do the process deployment via web service call
        new ServiceClientUtil().send(root, MIDDLEWARE_HOST + "/ode/services/DeploymentService");
}
```

As shown in the above `deploy` method, we process the following:

BPEL Validation: as usual and a good practice, we firstly validate the given
input prior to any further processing. Particularly, we check if the given crowd
work plan has been finalized and transformed into a BPEL artifact. If so, we
retrieve its byte content for further deploy preparation.

Deploy Preparation: as mentioned earlier, *Apache ODE* provides BPEL devel-
opers with a set of process management APIs - i.e. `pmapi` in the above code
fragment. As such, we use part of them, e.g. `OMElement` and `OMText`, to
encapsulate our BPEL artifact content to be ready for deployment.

Deploy Web Service Invocation: with the above prepared BPEL payload, we
do the process deployment through calling a built-in web service in *Apache
ODE*, whose endpoint is `/ode/services/DeploymentService`.

After deployment, we auto-start the deployed process with the initial input from
our requester when (s)he kicks off the execution phase. To this end, our `Runtime
Manager` realises the following.

```
public void start(String processName, String ...initialInput) {
    // validate arguments and construct the deployed BPEL process URL
    checkArgument(isNotBlank(processName), "Process name must exist!");
    checkArgument(initialInput != null && initialInput.length > 0, "Process input must be ready!");
    String processUrl = MIDDLEWARE_HOST + "/ode/processes/" + processName;
    // construct a SOAP request for BPEL web service invocation
    String initialRequest = initProcessRequest(processUrl + "?wsdl", initialInput);
    InputStream processInput = new ByteArrayInputStream(initialRequest.getBytes());
    // start the BPEL process via Apache ODE axis2 API
    HttpSoapSender.doSend(new URL(processUrl), processInput, null, 0, null, null, null);
}
```

As shown in the above `start` method, we process the following:

Initial Process Request Preparation: given a BPEL process exposes itself
as a web service for invocation, we then need to construct an initial SOAP
request to start it. To do so, we need its web service definition - i.e. WSDL
specification, and its initialization payload. As to the former, we can achieve
it through appending `?wsdl` suffix to the process endpoint. As to the latter,
we construct the payload with the passed `initialInput` argument.

Process Start web Service Invocation: with the above prepared process end-
point and the initial SOAP request, we then start the deployed BPEL process
through hitting its invocation web service via *Apache ODE axis2* API, e.g.
`HttpSoapSender`.

7.2.3   RequesterClient Web Application

RequesterClient is an end-user oriented web application facilitating human requesters
to participate in advertising a crowd task or starting the crowdsourcing execution.
As much of the implementation concerns and event communication mechanisms
are similar to the other web-based module, we do not show the details. One of the
important function of this application though is to finalise the work plan and publish
it to *CoordinationMiddleware* . When the requester selects the final plan, an event
`PLAN_SELECTED` is sent to *CoordinationMiddleware* through the same uniform event-
handling service. *CoordinationMiddleware* then will ask the corresponding event
handler to retrieve the final plan from the passed event and transform it into a BPEL
artifact as explained earlier. Fig. 7.11 shows a screen capture of the see a successful
BPEL generation log on *CoordinationMiddleware* after the `PLAN_SELECTED` event.

When the requester kicks off the execution of a crowd task plan through the front end

```
GDL2BPEL
Generate WSDL                                      ⬅  WSDL
Generate WSDL definitions                             generator
Generate WSDL types
Generate WSDL message
Generate WSDL portType
Generate WSDL binding
Generate WSDL service
Generate WSDL import
Generate WSDL plnk
combine all wsdl elements
Generate WSDL done                                 ⬅  BPEL
Begin to Generate BPEL                                elements
Generate BPEL process                                generator
Generate BPEL import
Generate BPEL partners                                BPEL
Generate BPEL variables                               control-flow
Generate BPEL sequence                             ⬅  generator
generateMainProcess is called
Generate total BPEL
Generate BPEL done
```
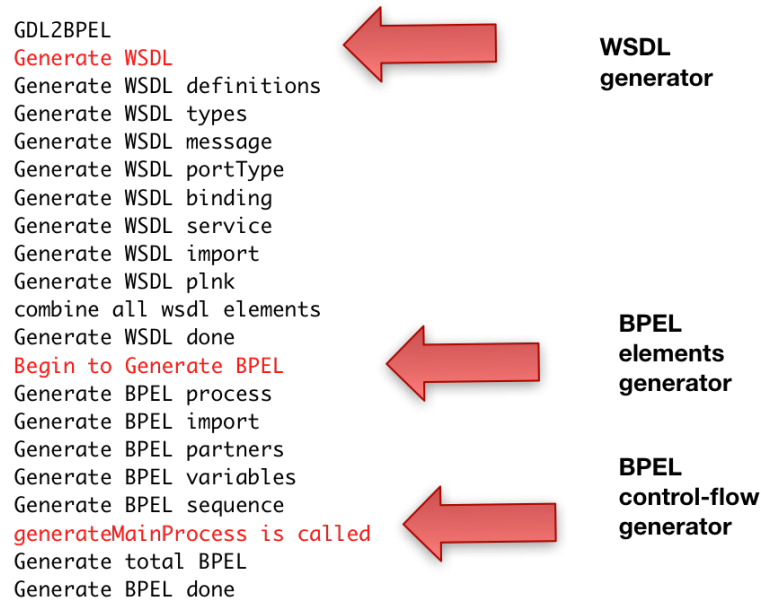
Figure 7.11: BPEL generation process

of `Execution Manager`, its back end will handle the process in the `startExecution` method (shown in Listing 7.8).

Listing 7.8: Starting the Execution of Chosen Plan
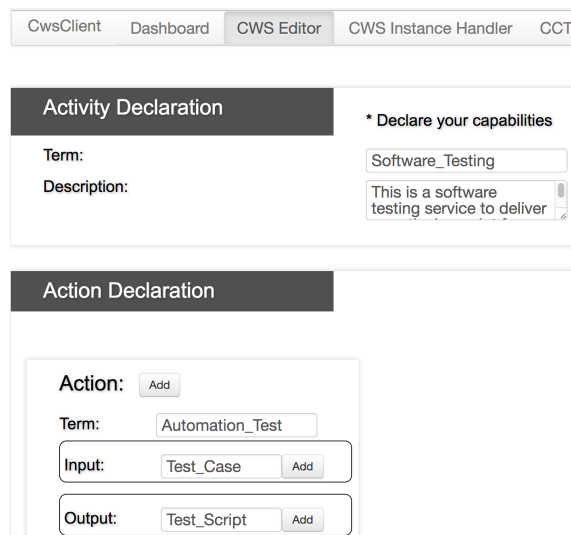
```java
@Path("/requester/execution/{planId}/kickoff")
@POST
@Consumes(APPLICATION_JSON)
public Response startExecution(@PathParam("planId") long planId, List<HumanArtifact> input) {
    try {
        // input validation
        checkArgument(planId > 0, "Invalid plan ID!");
        checkArgument(isNotEmpty(input), "Initial input must be ready!");
        // prepare 'EXECUTION_STARTED' event and call Middleware for update
        ExecutionStartedEvent event = prepareEvent(EXECUTION_STARTED, planId, input);
        middleware.handleEvent(event);
        // redirect the front end to requester's dashboard
        return Response.seeOther(new URI("/requester/dashboard")).build();
    } catch (Exception e) {
        return Response.serverError().build();
    }
}
```

After ensuring the initial input, **ExecutionStartedEvent** is sent to *Coordination-Middleware* and the response directs the human requester to his/her dashboard.

7.3 Use Case Demonstration

In this section, we use the motivating scenario in Sec. 1.1, i.e. *software testing process crowdsourcing*, as a scenario to do a walk-through demonstration of how modules work together in our complex task crowdsourcing context.

7.3.1 Creating and Publishing a Human Service

Figure 7.12: CWS definition through CWS Editor

To start with, by using `CWS Editor` as shown in Fig. 7.12, a tester declares his software testing capabilities. He defines a `Software_Testing` activity and its concrete `Automation_Test` action. It is noted that, in our prototype implementation, the terms are derived from a lookup list of domain specific term database (i.e., simple dictionary of software testing domain) that we maintain internally. The concrete `Automation_Test` action itself has input and output, `Test_Case` and `Test_Script` artifacts, respectively.

As a result of editing these details, the tester tells SOC4Crowd that, in his *software testing* service, an *automation test* action will be conducted to accept a test case

instruction and generate a script for test automation. With other details collected from the editor, it will instantiate the human activity model and generate a WSDL artifact as a CWS interface at both human and machine levels, respectively.

Figure 7.13: CWS publication through CWS Editor

After defining the *software testing* service, the tester needs to publish it for discovery. To do so, he can describe the service context through another workspace in `CWS Editor` as shown in Fig. 7.13. Through this, he can upload his *software quality assurance analyst* profile as a *software testing* service provider (e.g. concrete testing skill set, the past testing project experience). Then he can specify the availability and cost of the *software testing* service. `CWS Editor` will update CWS data model and register the new created *software testing* CWS in *CoordinationMiddleware* for discovery.

In this scenario, we assume that there are many other software testing related services defined and published this way in the platform.

7.3.2 Two-phased Complex Task Crowdsourcing

When the requester crowdsources a *software testing process* through our *SOC4Crowd* framework, she will go through the following steps.

Step 1: Complex Crowd Task Advertising

A new crowdsourcing process starts from the requester advertising her complex crowd task.

As shown in Fig. 7.14, the requester populates the details of her *software testing process* task prior to advertising it to the crowd. Specifically, she starts with giving it a concrete task `name` - i.e. *RMS testing process*, along with a further `description` that highlights its task goal - i.e. *a software acceptance test on their web application before delivering it to their client*. Then she specifies the `deadline` and `budget` constraints of completing this task. To draw attention of crowd workers in software development domain, she `tags` this task using *Software_Testing* and *Quality_Assurance* terms from a lookup list.

Figure 7.14: Advertising a complex crowd task in RequesterClient Application

Similarly, she uses other terms to specify the input and expected output to define the `workload` of current task, e.g. *Business_Requirement_Specification* as input and *Test_Plan_Specification* as output. It is worth noting that, for each input, the requester also needs to specify its `value` - i.e. the web location of current input ar-

tifact. Yet, for each output, she only needs to define the type of this output artifact using a proper term, as the current output content has not been produced yet.

A CCTS data model is initiated and a `TASK_ADVERTISED` event is sent to *Coordina-tionMiddleware* who broadcasts the *RMS testing process* task. Subsequently, crowd workers can see it on their *CwcClient* dashboard (see Fig. 7.15), and participate in its planning stage.

Figure 7.15: Advertised complex crowd tasks on CWS Dashboard

Step 2: Plan Phase – Task Decomposition and Plan Selection

In this step, the original *RMS Testing Process* task will be decomposed by the crowd workers to generate multiple candidate plans and the requester will choose one of them as the final working plan for the later execution.

Figure 7.16: Original crowd task details prior to decomposition

Task Decomposition by Crowd Workers: From the *CwsClient* dashboard, participating crowd workers can initiate the `CCTS Editor` for task decomposition by following the `Participate` action link. As shown in Fig. 7.16, the crowd worker can firstly view the details of *RMS Testing Process* task before any operation. Based on those original information, the crowd worker can proceed with decomposition operations by specifying *AND* or *OR* operators, and the number of sub-tasks that the current task is divided into.
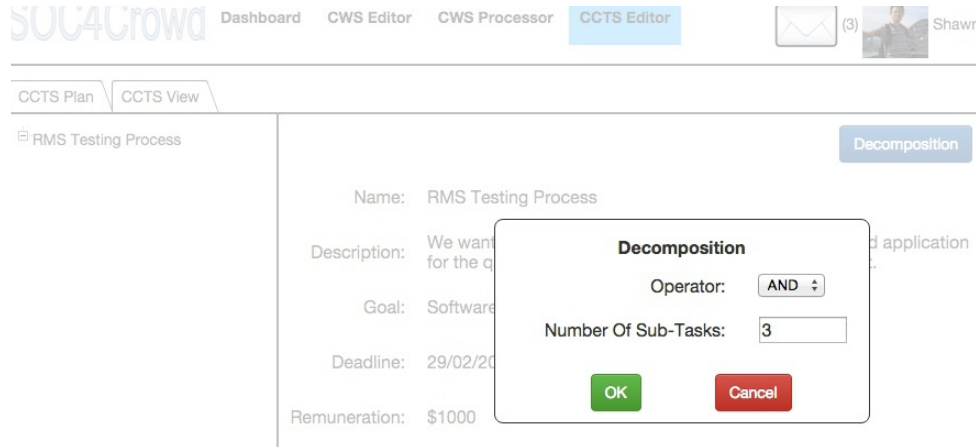

Figure 7.17: CCTS decomposition through CCTS Editor

Fig. 7.17 and Fig. 7.18 illustrate that the crowd worker decomposes the original *RMS Testing Process* task into three sub-tasks, i.e. *Test Design*, *Test Execution*, and *Test Report*, through the *AND* operator. Then the worker needs to further populate each sub-task details. For illustration, let us take *Test Design* sub-task.

Similar to its parent *RMS Testing Process*, it is specified with a concrete `name` and `description`, as well as a `deadline` and `cost` constraint. To better express its domain-specific meaning, the crowd worker `tags` it with *test_plan*, *test_design*, and *test_strategy* terms.

The input and output, of *Test Design* sub-task are specified. The names here need to match that of its respective parent and sibling tasks to establish a proper data flow
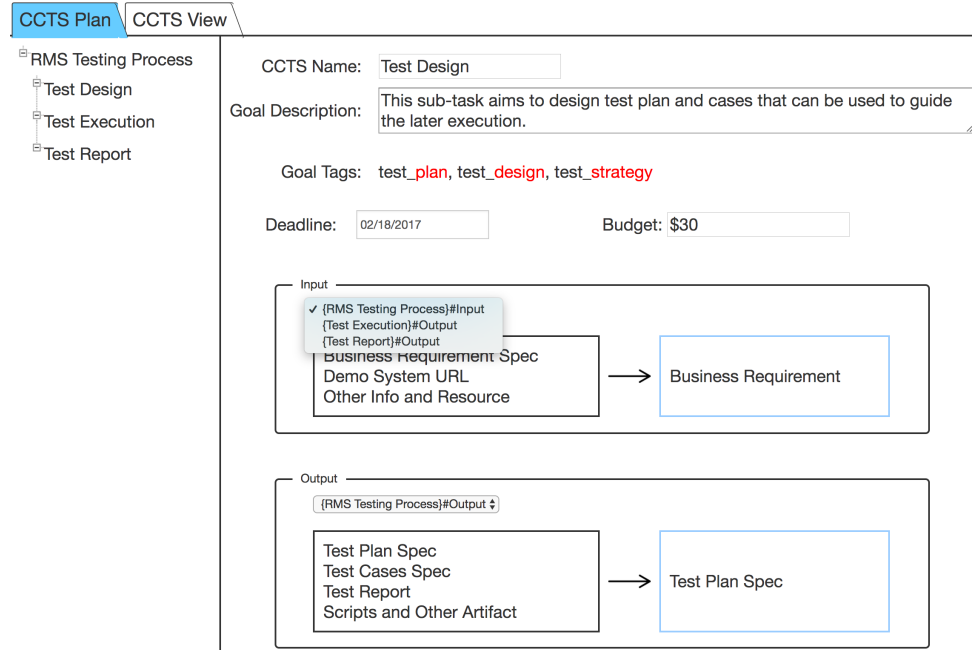
Figure 7.18: Sub-CCTS definition through CCTS Editor

for execution. Since the sub tasks are created with an *AND* operator, the input of *Test Design* comes from either the input of its parent - i.e. *RMS Testing Process*, or the output of its siblings - i.e. *Test Execution* and *Test Report*. Regarding its output, *Test Plan Spec* is selected as the output of *Test Design*. Through this workload definition, when *Test Design* sub-task is executed, its input will be instantiated with a *Business Requirement* artifact passed from its parent. When its output - i.e. *Test Plan Spec* artifact, is generated, it will be passed to its parent as part of the output of *RMS Testing Process*.

An example of a completed task decomposition can be seen in the tree-structured diagram in Fig. 7.19. After decomposition, the worker can submit it as a candidate plan. The `CCTS Editor` back end will send a `PLAN_SUBMITTED` event to *CoordinationMiddleware* to notify the requester about a a new work plan from the crowd.

Plan Selection by the Requester: When the requester has received a set of plans she can review each of those plans and select one of them as the final crowd work plan. As shown in Fig. 7.19, the requester has reviewed three different plans
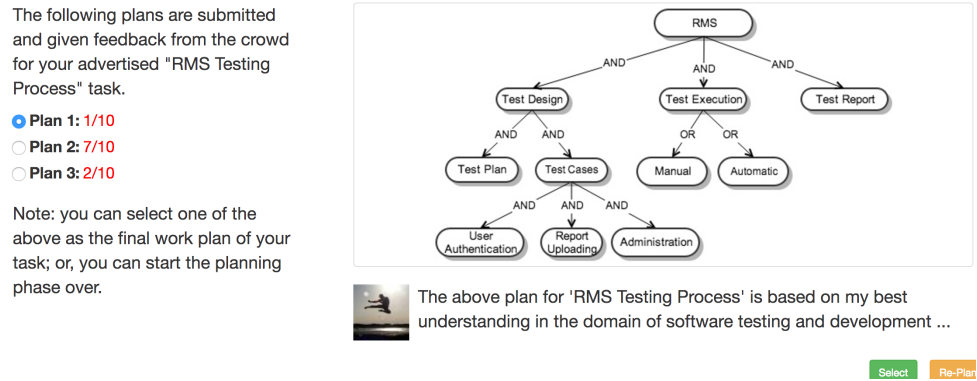

The following plans are submitted and given feedback from the crowd for your advertised "RMS Testing Process" task.

○ **Plan 1:** 1/10
○ **Plan 2:** 7/10
○ **Plan 3:** 2/10

Note: you can select one of the above as the final work plan of your task; or, you can start the planning phase over.

The above plan for 'RMS Testing Process' is based on my best understanding in the domain of software testing and development ...

Select   Re-Plan

Figure 7.19: Plan selection by the requester in RequesterClient

and chosen one. At this point, the back-end program of `Plan Manager` sends a `PLAN_SELECTED` event to *CoordinationMiddleware* who subsequently binds each leaf sub-task in the selected plan with a particular human service and transforms the selected plan into a BPEL artifact for the later execution phase.

Step 3: Execution Phase – Human Service Interaction

When the requester kicks off the crowdsourcing execution phase, the human services bound to the plan are called and orchestrated to complete the work. As such, the typical scenario in this phase is CWS interaction.

As designed and explained earlier, when a leaf sub-CCTS without any children nodes in its tree-structured plan comes to execution, its bound CWS gets requested and the target crowd worker gets notified by both an email and a dashboard message. For instance, when it comes to the *Manual Test* leaf sub-CCTS execution in our *RMS testing process* case, the *CoordinationMiddleware* firstly sends a request to its bound CWS whose corresponding human activity is *Software Testing* and the

Figure 7.20: Email notification in execution

concrete action is *Manual Testing*. This service request is in SOAP format and can be seen in Fig. 7.9. Then our `Interaction Protocol Manager` in *CwsClient* intercepts it and transforms it into both the email and dashboard notification, which can be seen in Fig. 7.20 and Fig. 7.21, respectively.
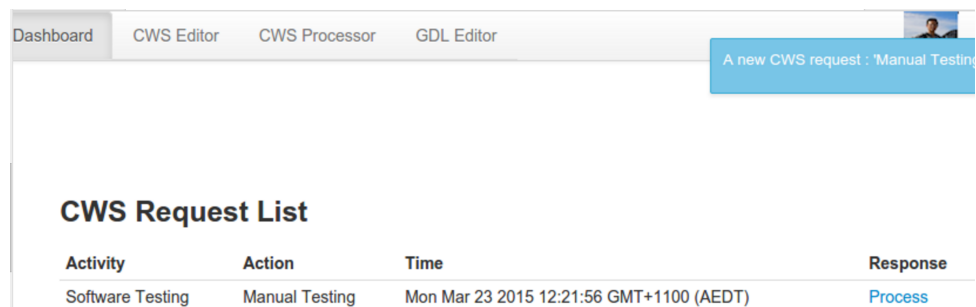
Figure 7.21: Dashboard notification during the crowdsourcing execution

When the crowd worker receives the above notification, as shown in Fig. 7.21, he can pick up that *Manual Testing* service request in the *CWS Request List* section of his or her dashboard via clicking the *Process* link. Then he will be directed to an user interface of our `CWS Instance Handler`, which is shown in Fig. 7.22, to view the request details, e.g. *Test Plan* input artifact.

With those visualized request information, the crowd worker then performs his or her

Figure 7.22: CWS instance handling

Manual Testing action outside of our system, e.g. logging into the testing environment of *RMS* web application and conducting the actual software testing process by following the instruction in the given *Test Plan* artifact. When he finishes the actual testing work and generates an *Issue Report* artifact as the output of *Manual Test* action, he can submit this output as his CWS response through the same user interface as above. Accordingly, our `Interaction Protocol Manager` converts this human service response into a callback SOAP message and sends it to *CoordinationMiddleware* for correlation. At this point, a CWS interaction is completed. Further, our *CoordinationMiddleware* would continue initiating another CWS interaction based on the BPEL specification, until the whole execution has been finished.

As we can see from the above *RMS testing process* use case walk-through, our proposed concepts and prototypical implementation of *SOC4Crowd* demonstrate how the complex task crowdsourcing can be realised through existing technologies. Next, we will summarize our research and highlight the future work.

Chapter 8

Conclusion and Future Work

In this thesis, we have firstly introduced the emerging trend in the crowdsourcing field - i.e. **crowdsourcing the complex task**. In order to support that in a systematic manner, we then identified a key feature - i.e. *coordination*, which is missing in most of current crowdsourcing platforms and other related work. Therefore, this thesis aims to bring coordination support into the crowdsourcing domain from service-oriented computing perspective, after seeing the similarity between crowdsourcing and SOC, and being motivated by the potential of applying the mature service interaction and orchestration technologies into human interaction and coordination.

To that end, first of all, we proposed a conceptual framework to re-design three key elements in the online crowdsourcing platforms from SOC standpoint. That is, i). *Crowd Worker*: in our framework, we proposed a concept called **Crowd Workers as Services (CWS)** in which each different, distributed crowd worker is abstracted into a similar service-oriented profile. In doing so, they can be better described, easier discovered and composed into the complex crowd work through the mature SOC technologies, when compared with a simple self-introduction and

the historical performance rating as the worker representation in most of current crowdsourcing platforms. ii). *Crowd Task*: in our work, we proposed another concept called **Complex Crowd Task as a Schema (CCTS)** in which we define the complex crowd work outsourced by the requester using a workflow-based schema. This schema consists of a set of units of work - i.e. atomic sub-tasks, along with their inter-dependencies, which defines the complex crowd task more clearly when compared with the simple task description in most of current crowdsourcing platforms. More importantly, through those atomic sub-tasks and their dependencies, this schema provides us with a guideline for the complex crowdsourcing coordination and management. iii). *Crowd Work and Participants Management*: when compared with the one-off task and worker matching process in most of current crowdsourcing platforms, we proposed a more comprehensive **Coordination Protocol** in our work to define and refine the complex crowd task, find and bind right crowd workers with right atomic tasks, coordinate their work and request their services to produce content, monitor and manage their performance and the quality of their work.

By following this conceptual framework, we then presented its technical design and implementation - *SOC4Crowd*, as a prototype realising a subset of those proposed concepts. To realise our *CWS* concept, we firstly designed a human activity model to abstract human capabilities; then with a *Activity-to-WSDL* mapping mechanism, we encapsulated those human activities or capabilities into a web service format and exposed it to the public as human services for the later interaction and coordination. To realise our *CCTS* concept, apart from detailing its *goal*, *workload*, and *constraint* properties, we designed two decomposition operators - i.e. *AND* and *OR*, to create a set of sub-CCTS and define the data-flow dependencies among them. After realising both *CWS* and *CCTS* concepts as the data model in our *SOC4Crowd* system, we designed a *Two-Phase Crowdsourcing Management* as an operation model to realise the coordination support. During the *plan phase*, with the *human-level planning* design, the requester is able to crowdsource the working plan on his/her advertised

task to get a better refined task definition - i.e. a full *CCTS* schema, before execution. Further, with the *machine-level planning* design, we bind that schema with multiple *CWS* and make it executable for the later auto-coordination. During the *execution phase*, with the *coordination model and interaction protocol* design, our *SOC4Crowd* can coordinate and manage the interactions between the requester and crowd workers through the web service interaction and orchestration.

As an initial step of bringing the coordination and management support into the complex work crowdsourcing area, this thesis proposed a service-oriented framework mainly focusing on an end-to-end solution to allow both the requester and crowd workers to interact with each other during the whole complex crowdsourcing lifecycle. Therefore, some of its detailed stages or steps need to be improved. For instance, the current *matching* between CCTS and CWS is realised through the tagging approach, which is not ideal enough as it requires a quite amount of effort on maintaining the internal domain-specific terms. To make it better in the future, we could take the semantics into account.

Bibliography

[AB14]   Xu Anbang and Brian P Bailey. A system for receiving crowd feedback
           on visual designs. In *Proceedings of the companion publication of the
           17th ACM conference on Computer supported cooperative work & social
           computing - CSCW Companion '14*, 2014.

[ABSK11]   Kittur Aniket, Smus Boris, Khamkar Susheel, and Robert E Kraut.
           CrowdForge. In *Proceedings of the 24th annual ACM symposium on
           User interface software and technology - UIST '11*, 2011.

[AHDB15]   Xu Anbang, Rao Huaming, Steven P Dow, and Brian P Bailey. A
           classroom study of using crowd feedback in the iterative design process.
           In *Proceedings of the 18th ACM Conference on Computer Supported
           Cooperative Work & Social Computing - CSCW '15*, 2015.

[AMB12]   Kulkarni Anand, Can Matthew, and Hartmann Björn. Collaboratively
           crowdsourcing workflows with turkomatic. In *Proceedings of the ACM
           2012 conference on Computer Supported Cooperative Work - CSCW
           '12*, 2012.

[ANM⁺13]   Kittur Aniket, Jeffrey V Nickerson, Bernstein Michael, Gerber Eliza-
           beth, Shaw Aaron, Zimmerman John, Lease Matt, and Horton John.
           The future of crowd work. In *Proceedings of the 2013 conference on
           Computer supported cooperative work - CSCW '13*, 2013.

[ASPR12]   Kittur Aniket, Khamkar Susheel, André Paul, and Kraut Robert.
           CrowdWeaver. In *Proceedings of the ACM 2012 conference on Com-
           puter Supported Cooperative Work - CSCW '12*, 2012.

[BBA12]   Meriem Benhaddi, Karim Baïna, and El Hassan Abdelwahed. A user-
           centric mashuped soa. *IJWS*, 1:204–223, 2012.

[BLS14]   Paul Belleflamme, Thomas Lambert, and Armin Schwienbacher.
           Crowdfunding: Tapping the right crowd. *Journal of business ven-
           turing*, 29(5):585–609, 2014.

[Bra08]   Daren C Brabham. Crowdsourcing as a model for problem solving: An introduction and cases. *Convergence*, 14(1):75–90, 2008.

[CED14]   Preist Chris, Massung Elaine, and Coyle David. Competing or aiming to be average? In *Proceedings of the 17th ACM conference on Computer supported cooperative work & social computing - CSCW '14*, 2014.

[DFS12]   Schall Daniel, Skopik Florian, and Dustdar Schahram. Expert discovery and interactions in mixed Service-Oriented systems. *IEEE Trans. Serv. Comput.*, 5(2):233–245, 2012.

[DS05]   Schahram Dustdar and Wolfgang Schreiner. A survey on web services composition. *International Journal of Web and Grid Services*, 1(1):1–30, 2005.

[DSA$^+$14]   Retelny Daniela, Robaszkiewicz Sébastien, To Alexandra, Walter S Lasecki, Patel Jay, Rahmati Negar, Doshi Tulsee, Valentine Melissa, and Michael S Bernstein. Expert crowdsourcing with flash teams. In *Proceedings of the 27th annual ACM symposium on User interface software and technology - UIST '14*, 2014.

[DSBB10]   Schall Daniel, Dustdar Schahram, and M Brian Blake. Programming human and Software-Based web services. *Computer*, 43(7):82–85, 2010.

[ECRSS16]   L Elisa Celis, Sai Praneeth Reddy, Ishaan Preet Singh, and Vaya Shailesh. Assignment techniques for crowdsourcing sensitive tasks. In *Proceedings of the 19th ACM Conference on Computer-Supported Cooperative Work & Social Computing - CSCW '16*, 2016.

[EJMG15]   Harburg Emily, Hui Julie, Greenberg Michael, and Elizabeth M Gerber. Understanding the effects of crowdfunding on entrepreneurial Self-Efficacy. In *Proceedings of the 18th ACM Conference on Computer Supported Cooperative Work & Social Computing - CSCW '15*, 2015.

[ELMG16]   Agapie Elena, Colusso Lucas, Sean A Munson, and Hsieh Gary. PlanSourcing: Generating behavior change plans with friends and crowds. In *Proceedings of the 19th ACM Conference on Computer-Supported Cooperative Work & Social Computing - CSCW '16*, 2016.

[ERC14]   Pavlick Ellie, Yan Rui, and Callison-Burch Chris. Crowdsourcing for grammatical error correction. In *Proceedings of the companion publication of the 17th ACM conference on Computer supported cooperative work & social computing - CSCW Companion '14*, 2014.

[FFG+14]   A Facoetti, S Franceschini, O Gaggi, G Galiazzo, S Gori, CE Palazzi, and M Ruffino. Multiplatform games for dyslexia identification in preschoolers. In *Consumer Communications and Networking Conference (CCNC), 2014 IEEE 11th*, pages 1152–1153. IEEE, 2014.

[FPO+11]   Raymond P Fisk, Lia Patrício, Andrea Ordanini, Lucia Miceli, Marta Pizzetti, and A Parasuraman. Crowd-funding: transforming customers into investors through innovative service platforms. *Journal of service management*, 22(4):443–470, 2011.

[Har15]   Christopher G Harris. The effects of Pay-to-Quit incentives on crowd-worker task quality. In *Proceedings of the 18th ACM Conference on Computer Supported Cooperative Work & Social Computing - CSCW '15*, 2015.

[HG15]   Julie S Hui and Elizabeth M Gerber. Crowdfunding science. In *Proceedings of the 18th ACM Conference on Computer Supported Cooperative Work & Social Computing - CSCW '15*, 2015.

[How06]   Jeff Howe. The rise of crowdsourcing. *Wired magazine*, 14(6):1–4, 2006.

[HT11]   Joseph M. Hellerstein and David L. Tennenhouse. Searching for jim gray: A technical overview. *Commun. ACM*, 54(7):77–87, July 2011.

[JSD16]   Chan Joel, Dang Steven, and Steven P Dow. Improving crowd innovation with expert facilitation. In *Proceedings of the 19th ACM Conference on Computer-Supported Cooperative Work & Social Computing - CSCW '16*, 2016.

[JTB15]   Thebault-Spieker Jacob, Loren G Terveen, and Hecht Brent. Avoiding the south side and the suburbs. In *Proceedings of the 18th ACM Conference on Computer Supported Cooperative Work & Social Computing - CSCW '15*, 2015.

[JWR15]   Solomon Jacob, Ma Wenjuan, and Wash Rick. Don't wait! In *Proceedings of the 18th ACM Conference on Computer Supported Cooperative Work & Social Computing - CSCW '15*, 2015.

[KJLW+15]   Luther Kurt, Tolentino Jari-Lee, Wu Wei, Pavel Amy, Brian P Bailey, Agrawala Maneesh, Hartmann Björn, and Steven P Dow. Structuring, aggregating, and evaluating crowdsourced design critique. In *Proceedings of the 18th ACM Conference on Computer Supported Cooperative Work & Social Computing - CSCW '15*, 2015.

[KMB+15]   Zyskowski Kathryn, Meredith Ringel Morris, Jeffrey P Bigham, Mary L Gray, and Shaun K Kane. Accessible crowdwork? In *Proceedings of the 18th ACM Conference on Computer Supported Cooperative Work & Social Computing - CSCW '15*, 2015.

[LAK16]   Yu Lixiu, Kittur Aniket, and Robert E Kraut. Encouraging "outside-the- box" thinking in crowd innovation through identifying domains of expertise. In *Proceedings of the 19th ACM Conference on Computer-Supported Cooperative Work & Social Computing - CSCW '16*, 2016.

[LKA16]   Yu Lixiu, Robert E Kraut, and Kittur Aniket. Distributed analogical idea generation with multiple constraints. In *Proceedings of the 19th ACM Conference on Computer-Supported Cooperative Work & Social Computing - CSCW '16*, 2016.

[MMJ+16]   Muller Michael, Keough Mary, Wafer John, Geyer Werner, Alberto Alvarez Saez, Leip David, and Viktorov Cara. Social ties in organizational crowdfunding: Benefits of Team-Authored proposals. In *Proceedings of the 19th ACM Conference on Computer-Supported Cooperative Work & Social Computing - CSCW '16*, 2016.

[MOS+14]   Cefkin Melissa, Anya Obinna, Dill Steve, Moore Robert, Stucky Susan, and Omokaro Osariemo. Back to the future of organizational work. In *Proceedings of the companion publication of the 17th ACM conference on Computer supported cooperative work & social computing - CSCW Companion '14*, 2014.

[MST+15]   Kobayashi Masatomo, Arita Shoma, Itoko Toshinari, Saito Shin, and Takagi Hironobu. Motivating Multi-Generational crowd workers in Social-Purpose work. In *Proceedings of the 18th ACM Conference on Computer Supported Cooperative Work & Social Computing - CSCW '15*, 2015.

[Nor15]   Karen Northon. NASA Uses Crowdsourcing for Open Innovation Contracts, 2015.

[Obi15]   Anya Obinna. Bridge the gap! In *Proceedings of the 18th ACM Conference on Computer Supported Cooperative Work & Social Computing - CSCW '15*, 2015.

[PFSM16a] Kucherbaev Pavel, Daniel Florian, Tranquillini Stefano, and Marchese Maurizio. Crowdsourcing processes: A survey of approaches and opportunities. *IEEE Internet Comput.*, 20(2):50–56, 2016.

[PFSM16b]   Kucherbaev Pavel, Daniel Florian, Tranquillini Stefano, and March-ese Maurizio. ReLauncher: Crowdsourcing Micro-Tasks runtime con-troller. In *Proceedings of the 19th ACM Conference on Computer-Supported Cooperative Work & Social Computing - CSCW '16*, 2016.

[PKLSH13]   Cheong Ha Park, Son KyoungHee, Joon Hyub Lee, and Bae Seok-Hyung. Crowd vs. crowd. In *Proceedings of the 2013 conference on Computer supported cooperative work - CSCW '13*, 2013.

[PRM10]   C. E. Palazzi, M. Roccetti, and G. Marfia. Realizing the unex-ploited potential of games on serious challenges. *Comput. Entertain.*, 8(4):23:1–23:4, December 2010.

[PTDL07]   Michael P Papazoglou, Paolo Traverso, Schahram Dustdar, and Frank Leymann. Service-oriented computing: State of the art and research challenges. *Computer*, 40(11), 2007.

[Ran03]   Shuping Ran. A model for web services discovery with qos. *SIGecom Exchanges*, 4(1):1–10, March 2003.

[RH16]   Hope Reese and Nick Heath. Inside Amazon's clickworker platform: How half a million people are being paid pennies to train AI, 2016.

[RS05]   Jinghai Rao and Xiaomeng Su. *A Survey of Automated Web Service Composition Methods*, pages 43–54. Springer Berlin Heidelberg, Berlin, Heidelberg, 2005.

[RYMOV12]   Vaculin Roman, Chee Yi-Min, Daniel V Oppenheim, and Lav R Varsh-ney. Work as a service meta-model and protocol for adjustable visibil-ity, coordination, and control. In *2012 Annual SRII Global Conference*, 2012.

[SFPF15]   Tranquillini Stefano, Daniel Florian, Kucherbaev Pavel, and Casati Fabio. Modeling, enacting, and integrating custom crowdsourcing pro-cesses. *ACM Transactions on the Web*, 9(2):1–43, 2015.

[SJ16]   Kairam Sanjay and Heer Jeffrey. Parting crowds: Characterizing di-vergent interpretations in crowdsourced annotation tasks. In *Proceed-ings of the 19th ACM Conference on Computer-Supported Cooperative Work & Social Computing - CSCW '16*, 2016.

[SZX+15]   Cheng Shiwei, Sun Zhiqiang, Ma Xiaojuan, Jodi L Forlizzi, Scott E Hudson, and Dey Anind. Social eye tracking. In *Proceedings of the 18th ACM Conference on Computer Supported Cooperative Work & Social Computing - CSCW '15*, 2015.

[TASF+16] Kandappu Thivya, Misra Archan, Cheng Shih-Fen, Jaiman Nikita, Tandriansyah Randy, Chen Cen, Hoong Chuin Lau, Chander Deepthi, and Dasgupta Koustuv. Campus-Scale mobile Crowd-Tasking: Deployment & behavioral insights. In *Proceedings of the 19th ACM Conference on Computer-Supported Cooperative Work & Social Computing - CSCW '16*, 2016.

[VDAVH04] Wil Van Der Aalst and Kees Max Van Hee. *Workflow management: models, methods, and systems*. MIT press, 2004.

[Vog07] Werner Vogels. Help Find Jim Gray, 2007.

[WPB13] Ingo Weber, Hye-Young Paik, and Boualem Benatallah. Form-based web service composition for domain experts. *ACM Trans. Web*, 8(1):2:1–2:40, 2013.

[YWZ+04] Kun Yue, Xiao-Ling Wang, Ao-Ying Zhou, et al. Underlying techniques for web services: A survey. *Journal of Software*, 15(3):428–442, 2004.