ExCovery – A Framework for Distributed System Experiments and a Case Study of Service Discovery

Andreas Dittrich*, Stefan Wanja† and Miroslaw Malek*
* Università della Svizzera italiana (USI), Lugano, Switzerland
Email: {andreas.dittrich,miroslaw.malek}@usi.ch
† DSPECIALISTS Digitale Audio- und Messsysteme GmbH, Berlin, Germany
Email: post@stefan-wanja.de

***Abstract*—Experiments are a fundamental part of science. They are needed when the system under evaluation is too complex to be analytically described and they serve to empirically validate hypotheses. This work presents the experimentation framework *ExCovery* for dependability analysis of distributed processes. It provides concepts that cover the description, execution, measurement and storage of experiments. These concepts foster transparency and repeatability of experiments for further sharing and comparison. *ExCovery* has been tried and refined in a manifold of dependability related experiments during the last two years. A case study is provided to describe service discovery as experiment process. A working prototype for IP networks runs on the Distributed Embedded System (DES) wireless testbed at the Freie Universität Berlin.**

I. Introduction

Experiments are a fundamental part of science. They are needed when the system under evaluation is too complex to be analytically described and they serve to empirically validate hypotheses. This is especially true for dependability analyses in distributed systems, which are often of extensive size and exhibit complex fault behavior. Experiments are fundamental to support research in this area. However, due to their diverse focuses it remains difficult to repeat, classify, evaluate and compare the different results. A consistent experimentation environment (EE) could help to unify related experiments and thus, greatly improve the impact of individual results. In this work, we present *ExCovery*, an EE for dependability research of distributed processes. A formal description to specify experiments has been developed, which forms the basis of *ExCovery*. It allows for automatic checking, execution and additional features, such as visualisation of experiments. *ExCovery* is expected to foster repeatability and transparency by offering a unified experiment description, measurement mechanism and storage of results.

Service oriented architecture (SOA) describes services as the building blocks of system design. A service is an abstract functionality in a network provided by an interface clients can connect to. SOA enforces the principle of discoverability, which means that structured data is added to service descriptions to effectively publish and discover individual providers. Communication of this data is done using service discovery (SD). We provide a case study of using *ExCovery* for experiments on SD in wireless IP networks.

The rest of the paper is structured as follows. Section II covers the topics of scientific experimentation and design of experiments. Section III contains background information about service discovery. *ExCovery* is presented in Section IV, its concepts illustrated with examplary experiment description code. The description of SD as a specific experiment process follows in Section V. An overview of the current *ExCovery* prototype implementation is given in Section VI. Section VII concludes the work.

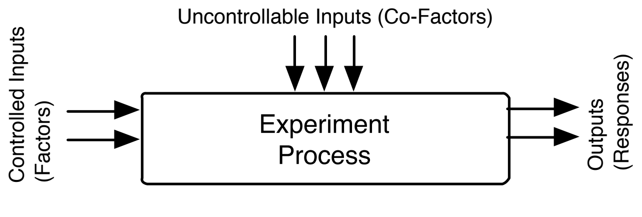
Uncontrollable Inputs (Co-Factors)
Controlled Inputs (Factors)
Experiment Process
Outputs (Responses)

Fig. 1. Model of a generic experiment process

II. The Art of Experimentation

Experiments play an important role in computer science by supporting theories inferred from observations or mathematical models. With increasing complexity of computer systems and networks, exploratory experiments are themselves the source of such theories. The subject of an experiment can be characterized as a black box process as illustrated in Figure 1. Inputs or factors can be controlled, outputs or responses observed. During an experiment it needs to be identified which factors exist and how they influence the responses. Controlling factors to observe their influence can be done one at a time or by manipulating multiple factors in a factorial experiment. Usually, experiments are run in series to capture the variation among multiple runs of the same experiment. Such series of controlled experiments can be called an experimental system [1]. Experiments need to be reliable in a sense that they must be verifiable when repeated under similar conditions. Experiment design must therefore keep repetition in mind and its publication must contain all necessary information to do so. Experiments further need to fulfill requirements for internal and external validity. Internal validity means that the causal relationships between the factors and responses should be verified. External validity deals with the generalization of experiment results.

A. Basic Terms and Principles

In the context of this work, an *experimentation environment* (EE) is defined as a set of tools with the purpose of describing, executing and evaluating experiments on a given subject, using a methodology specific to that subject. The actual setting in which the experiments and the EE are run is called *platform*.

1) Experiment Factors: There are different sources of variation among individual experiments which are called *factors*. A *treatment factor* is "any substance or item whose effect on the data is to be studied" [2, p.8]. A treatment factor can have continuous values but usually has discrete levels that are to be applied to study its effect. Treatment factors can be further classified as *design factors*, intentionally varied during the experiment, *held-constant factors*, whose impact is intentionally neglected and *allowed-to-vary factors*, known to have a minor influence that can be compensated by applying randomization and replication (see Section II-A3). [3, p.15] *Nuisance factors* on the other hand have an unwanted effect on the output. They can be divided into controllable and uncontrollable nuisance factors. The former, often called *blocking factors*, can be fixed by the experimenter to reduce their impact on the response. The latter, also called *covariates*, can not be set but measured. Their effects can be minimized by covariance analysis. A *noise factor* causes random variations in the responses but can be controlled during the experiment.

2) Experiment Design: As defined in [4], a *treatment* is the entire description of what can be applied to the treatment factors of an *experimental unit*, the smallest unit to which such treatment can be applied. An *observational unit* then is the smallest unit on which a response will be measured. The *experimental design* defines which treatments are to be observed on which experimental units [2, p.6]. Following [5], it can be divided into *treatment design*, a specification of the treatments used in an experiment, *error control design*, defining how the specified treatments are to be applied to reduce unwanted variations, and *sampling and observation design*, which decides on the observational units and whether uni- or multivariate observations are to be taken. Experiments tend to be time consuming. Proper planning of experiments follows a design to maximize the gained information per run, expressed in a higher precision of the response or in an increased significance of factor relationships. Improvements are achieved by properly structuring the factor and factor level variations over the required number of runs. A thorough explanation of experiment designs and for which objectives to apply them can be found in [2], [3].

3) Replication, Blocking and Randomization: To improve the validity of experiments, three interconnected principles are applied. *Replication* increases the number of experiment runs to be able to average out random errors in responses and to collect data about the variation in responses over a set of runs. Care has to be taken in the selection of held-constant factors to allow for proper replication. *Blocking* means partitioning observations into groups, or blocks, in such a way that observations in each block are collected under similar experimental conditions [2]. Statistical analysis requires that experiment observations are independently distributed random variables [3]. *Randomizing* the assignment of treatments to experimental units as well as the temporal order and spacial choice of multiple runs takes care of this requirement. An experiment design is called completely randomized when all treatment factors can be randomized.

B. Experiments in Computer Science

The role of experimentation in computer science has been the root of numerous debates, for example in [6], [7]. Seen as an engineering discipline, all subjects of inquiry in computer science are synthetic, as created by man. An often expressed opinion is that phenomena in computer science should be derived and explained following the construction of these subjects, instead of observing them as natural phenomena. However, the complexity of these subjects and their relationships prohibits such deduction. Additionally, the subjects of computer science have become part of the world around us and interact with it, creating not entirely synthetic hybrids. As such, observation as in natural sciences can be justified just as the testing of hypotheses to assess and understand man-made systems. When doing experiments in computer science, challenges exist mainly related to observation and repetition. It is not always clear which responses to observe and how to do it in such a way that the observation has no impact on the response itself. Modern computing systems provide ways to observe tens of thousands of parameters and observing them – measuring, recording and extracting – has to be done in the least invasive way. Also, repeating experiments can be difficult due to dependence on the original hardware platform. Where technology is proceeding at such rapid pace, experiments should be planned as independent as possible from the hardware they are running on, that is, if the hardware itself is not the subject of experimentation.

C. Experimentation Environment

To identify the demands addressed by *ExCovery* addresses, it is necessary to define what an EE is and what it needs to provide to an experimenter. In general, an EE allows to perform a certain class of experiments in a controlled environment. It facilitates the identification and manipulation of factors and the observation of these manipulations on the responses. The amount of possible control depends on the characteristics of the EE. For unwanted influences that cannot be controlled, the EE should provide the possibility to record them so they can be considered during analysis at a later stage. To foster the repeatability, correctness and transparency of experiments, an EE should use a description for setup, execution and evaluation of experiments. A common output format for measurements, logs and diverse meta information should be provided.

1) Network Experimentation Platforms: ExCovery focuses on experiments to evaluate the dependability of distributed processes, e.g. network protocols. To execute experiments and observe responses, it relies on network experimentation platforms. The most popular forms are simulators and testbeds, or mixed forms of both, such as virtualized testbeds or simulators with interfaces to real networks or real protocol implementations. It generally strengthens the external validity of an experiment if it is run in a diversity of platforms.

Simulators are software artifacts that simulate real-world processes by acting according to an abstract model of such a process. They can be discrete event-driven simulators, which calculate the state of the simulated object only when its state changes, or real-time simulators, which calculate the continuous behavior of the simulated object over time. Mixed forms exist, for example, where an event-driven simulator is synchronized to a wall clock. While simulators have a perfect reproducibility of experiments, good scalability and generally a reduced execution time, their abstractions often struggle to capture the properties and behavior of real-world distributed systems [8, p.2]. A testbed is a distributed system made of real network nodes. Testbeds usually provide means to manage experiment schedules and setup, data acquisition and storage. Testbeds allow less control over factors than simulators but measurements are the result of a realistic interplay of factors. As such, testbed usually allow to represent a specific environment (e.g. wireless mesh or large scale internet network) very well. An approach to unify generic network experimentation across simulators, emulators and testbeds is proposed with NEPI [9], an integration framework for network experimentation which creates a common model of experimentation that can be applied to many physical Testbeds as well as to the ns-3 simulator and the netns emulator. Another approach to achieve a unified system for executing distributed systems experiments on all kinds of testbeds is called Weevil [10], [11].

III. Service Discovery fundamentals

In SOA, emphasis is put on the consideration of different ownership domains, so an important aspect is the interoperability among services. There must be a way for potential partners to get to know of each other. This obligatory aspect is called visibility which is composed of awareness, willingness and reachability. Among the principles introduced by SOA to support its paradigm, discoverability is related to awareness. A comprehensive list and description of principles can be found in [12]. On the network side, service discovery (SD) focuses on awareness and discoverability: Structured data is added to services to be effectively published, discovered and interpreted. Service discovery protocols (SDPs) take care of communicating this data, to announce, enumerate and sort existing service instances.

A. *Basic Discovery Concepts and Roles*

An abstract service, also known as service type or service class, is provided by concrete service instances in the network. A set of service classes S can be provided on a set of providers P which then use an SD protocol to make the service known to interested users. SD typically connects the three different roles of user agent, service agent and directory agent [13]. They are also known as service consumer, service provider and service broker [14]. For the remainder of this paper, we will use the taxonomy of a general SD model developed by Dabrowski et al. [15], in which these roles are called service user (SU), service manager (SM) and service cache manager (SCM). An SM publishes its service on behalf of a service provider either autonomously or via an SCM. It makes a service description available with information on how and where its service can be invoked: The SM identity, a service type specification, an interface location or network address and optionally, various additional attributes. The SU discovers services on behalf of a user either by passively listening to announcements done by SAs or SCMs, by actively sending out queries to look for them, or by doing both. Discovery can happen in separate steps, enumerating discoverable instances first and then selectively retrieving the description. Also, not only services can be discovered, but administrative scopes, SCMs and service types, depending on the SDP. Finally, an SCM caches service descriptions of multiple SMs to maintain a list of present services that can be queried by SUs. SCMs are usually used to improve scalability. It should be noted that most SDPs implement also a local cache on SUs and SMs to reduce network load.

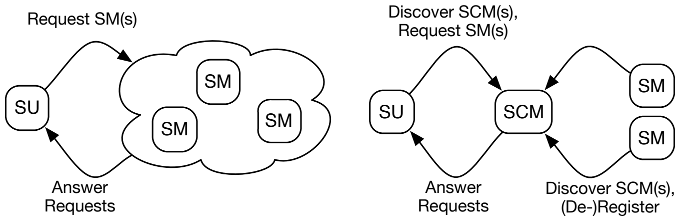
Request SM(s)
SU
SM
SM
SM
Answer Requests
Discover SCM(s), Request SM(s)
SU
SCM
SM
SM
Answer Requests
Discover SCM(s), (De-)Register

Fig. 2. Illustration of service discovery architectures: two-party (left) and three-party (right).

B. *Discovery Architecture and Communication*

Two different SD architectures can be distinguished, as depicted in Figure 2. In two-party or decentralized architecture, there exist only SUs and SMs in the network which communicate directly among each other. The architecture is called three-party or centralized if there is one or more SCM present. Centralized does not imply a preceding administratitive configuration because an SCM itself can be discovered at runtime as part of an SD process. There exist mixed forms that can switch among two- and three-party, called adaptive or hybrid architectures. Depending on the role and architecture, different communication types are used, unicast, multicast or broadcast. Some SDPs include routing mechanisms, hence, overlay networks in their communication logic which is generally called SD with structured communication approach. Others leave this to the underlying layers, following an unstructured approach. Furthermore, the communication scheme used for the actual discovery can be classified as passive (or lazy), active (or aggressive) or directed. In passive discovery, SUs discover discoverable items only by listening to their unsolicited announcements. When doing active discovery, SUs actively send out multi- or broadcast queries. In directed discovery, SUs actively send unicast queries to a given SCM or SM. There are many messages used by the SDPs to coordinate the distributed system, maintain a consistent state and optimize network traffic. The currently most common SDPs are presented and compared in [15]–[17]. In [18]–[20], SDPs for pervasive and ubiquitous computing systems are surveyed, which are the target platform of the prototype in Section VI.

IV. The Experimentation Environment *ExCovery*

We will now present the main concepts of the proposed experimentation environment *ExCovery* with its core, the formal abstract description of an experiment using the extensible markup language (XML). It includes definitions of the experiment with its input factors, the process to be examined, of fault injections or manipulations and diverse platform specific and informative declarations. *ExCovery* further provides a unified measurement concept that determines which and how data are stored for later analysis. An overview of the different concepts and the experiment work flow is illustrated in Figure 3.In the first preparation step, the experiment is designed by the experimenter following guidelines as mentioned in Section II-A2. The individual descriptions are explained in Section IV-C. Platform setup is necessary to prepare the translation of descriptions to the target platform, e.g. a specific testbed or simulator. Among others, this could include a deployment of programs, modules and configuration files. The experiment is then executed by the experiment master, a program that

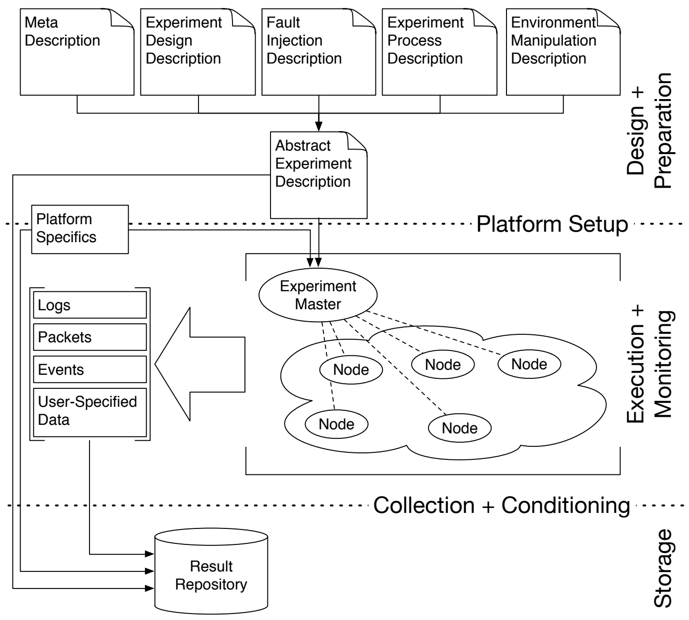

Fig. 3. Overview of *ExCovery* concepts and experiment workflow.

executes experiment runs as specified in the description. Each run is a sequence of actions performed on the participating nodes, described as the main process under evaluation and a set of injected faults or manipulations. The master and all nodes monitor and record dedicated parameters during each run, such as raw packet captures and the complete temporal sequence of actions and events. These data will be saved in a temporary location locally. After experiment execution, the collected data are collected and conditioned so that a common time base for all actions, events and packet measurements is established. Finally, data are stored into a single results database that contains all conditioned measurement data, created log files and the complete experiment plan with the exact sequence of treatments (see Sections IV-B and IV-F).

A. Platform Requirements

To integrate a specific target platform in *ExCovery*, it must support several features. Most of the features are needed to establish a controllable environment or to compensate for missing control and to allow detailed measurements. As such, these are mainly an issues for testbeds, simulators generally can be integrated with less effort.

1) Experiment Management: There must be a separate and reliable communication channel between the experiment master and the nodes participating in the experiment process. In simulators, this is usually provided by a software interface while testbeds need to possess physically separate and non-interfering network interfaces. During experiments, full, privileged and access to all nodes is mandatory. The platform needs to cleanly separate concerns of multiple users.

2) Connection Control: Full control over the network connections of the individual network nodes is needed. Network interfaces need to support activation and deactivation. Furthermore, it needs to be possible to manipulate packets sent over these interfaces based on defined rules. This covers dropping of packets, delaying, reordering, and modifying their content.

3) Measurement: There must be methods to capture packets with their exact local timestamps and their complete and unaltered content. To facilitate a comprehensive subsequent analysis, a packet tracking mechanism is required. Usually available in simulators, in testbeds this means tracking the routes of packets hop by hop, or attaching unique identifiers to packets [21]. Finally, the platform needs to support time synchronization among all participating nodes and it needs to support quantification of the synchronization error.

B. Measurement and Recording

The section clarifies the basic observations that are possible using *ExCovery*, how they can be observed and how this can help to unify related experiments. *ExCovery* follows the principle of collecting as much data as possible to support diverse analyses on the same experiment data at later time, emphasizing reusability and repeatability. Basic recordable data for network protocols are the results of protocol operations as reflected by state changes on the participants and network messages sent among participants. Additionally, *ExCovery* has a plugin concept to extend these data with custom measurements on demand.

1) Events: State changes on nodes in the context of *ExCovery* reflect *events* and occur, for example, when an experiment run is initialized or when a fault injection is started or stopped as defined by the experiment plan. Events are associated with the node on which they occur. They contain a local time stamp and may have additional parameters, such as the identifier of the experiment run that is being initialized. To control the experiment execution, nodes can be synchronized using global events (see Section IV-C2).

2) Packets: Packets are the basic communication data of network protocols. As opposed to events, single packets are not easily identified: Their location changes as they traverse the network, retransmissions and network loops complicate the correct localization at any given time. Packets are recorded to facilitate verification of the recorded event list and to derive statistical connection parameters during later analysis. A measured packet consists of a time stamp, representing the local occurrence of that packet, a unique identifier, a source and destination network address and the packet content itself.

3) Time: Events and packets have a local time stamp of to the node they were measured on. As *ExCovery* is focused on distributed systems, it defines mandatory measurements to be done before each run to estimate the time difference of each participant to a reference clock. This allows to construct a valid global time line of events and packets, avoiding causal conflicts due to local clocks deviating between experiment runs.

4) Topology: To improve repeatability, a rudimentary description of the network topology is measured as hop count between the participating nodes. This measurement is done before and after executing an experiment. A more advanced topology recording is anticipated for future versions of *ExCovery*.

5) Recording: Each participating node has its own temporary storage for recorded data, organized into data belonging to single runs and data valid for the complete experiment. Time synchronization measurements are stored on the experiment master. Plugins have a separate storage location on the node where the custom measurements are done. *ExCovery* does not impose a specific storage mechanism but requires that this storage is accessible during the subsequent collection and conditioning phase (see Section IV-F).

C. Abstract Experiment Description and Execution

ExCovery executes experiments on the base of an abstract description made up of three parts. The first contains the experiment design, which factors are applied in which combination and order. The second part contains manipulations on the process environment and the participants themselves, detailed in Section IV-D. The third part is the description of the distributed process to be examined. *ExCovery* uses XML to notate the description. An XML schema description is provided with the framework code. In the following, the parts that create an abstract description are defined and described.

Factor Part of the *treatment* applied to the *experimental unit*. Consists of a set of levels. Depending on the design, levels are applied one after another (OFAT) or randomized.
List of factors Contains all factors used, sorted. In an OFAT design the first factor varies least often during execution while the last factor changes every run.
Level Concrete value a factor can take, as input variables to the sub-processes of each run. Levels can be of different types. As such, they can control type and duration of fault injections (see Section IV-D) or represent mappings of abstract nodes to actors in the experiment process.
Set of levels All levels that should be applied during the experiment. Order of application is determined by the factor definition. There is only a single level if the factor should be kept constant during the whole experiment.
Replication factor Parameter defining an integer number of replications to be done with each treatment.
Abstract node Actor of the experiment process or of a node specific fault injector. Identified by a node identifier, such as a unique host name.
Environment node A node which is not participating as actor in any node specific process. Used for example to produce generate network load.
Actor description Process prototype to be executed on one specific actor of the experiment process. Each abstract node is mapped to one actor description, multiple abstract nodes can instantiate the same actor description.
Experiment process Actual experiment operation that is to be excuted and measured. Description consists of actions performed on multiple nodes, synchronized by flow control functions that wait for a certain time or for certain events issued by the nodes.
Manipulation process Main part of the *treatment*. Similar to experiment processes, represents a sequence of faults or impairments that should happen on a node.
Environment manipulation process As experiment process and manipulation process but not node specific. Uses similar synchronization methods, but controls manipulations to the environment, like traffic generation.

A rudimentary beginning of an experiment description is depicted in Figure 4. Two abstract nodes A and B are to be mapped by the processes described later. To support a basic classification of experiments, three parameters describing the discovery architecture and protocol used are defined as key-value pairs.

```
<experiment_name>"just a name"</experiment_name>
<totalnodes>"2"</totalnodes>
<abstractnodes>
  <abstractnode id="A"></abstractnode>
  <abstractnode id="B"></abstractnode>
</abstractnodes>
<sd_level>"user"</sd_level>
<sd_protocol>"zeroconf"</sd_protocol>
<sd_arch>"2-party"</sd_arch>
```

Fig. 4. Rudimentary experiment description with informative parameters about discovery process.

```
<factorlist>
  <factor id="fact_nodes" type="actor_node_map"
      usage="blocking">
    <levels><level>
    <actor id="actor0"><instance id="0">A</instance></actor>
    <actor id="actor1"><instance id="0">B</instance></actor>
    </level></levels>
  </factor>
  <factor usage="random" type="int" id="fact_pairs">
    <!-- number of randomly distributed node pairs to
         ↪ generate load -->
    <levels>
      <level>5</level><level>20</level>
    </levels>
  </factor>
  <factor usage="constant" id="fact_bw" type="int">
    <!-- datarate generated load</description> -->
    <levels>
      <level>10</level><level>50</level><level>100</level>
    </levels>
  </factor>
  <replicationfactor usage="replication" type="int"
      id="fact_replication_id">1000
  </replicationfactor>
</factorlist>
```

Fig. 5. Several defined factors in the description and their levels.

1) Execution: To execute the overall experiment and its individual runs from the abstract experiment description, *ExCovery* generates treatment plans from replications, the factors and their levels. Plans are OFAT if no custom factor level variation plan is given. The various random values used in *ExCovery* are generated using pseudo-random generators. This allows for perfect repeatability of random sequences used within an experiment when initialized with the same seed. Which seed is used for initialization is clearly defined in the experiment description so that all random sequences can be reproduced. *ExCovery* uses four internal functions for the experiment flow. Experiments are initialized by calling `experiment_init` on every participant, which takes care of the necessary preparations before all individual experiment runs. Each run is then initialized by `run_init`. There further exist the respective exit functions `run_exit` and `experiment_exit`.

Figure 5 shows the definition of several factors and their levels. First, the abstract nodes defined in Figure 4 are assigned actor roles $actor0$ and $actor1$. Then, two different factor are defined with various levels to describe the load generation that is to be applied during experiments, in this case a random number of 5 and 20 node pairs that will exchange data with first 10, 50 and then 100 kilobits per second. Each treatment will be repeated 1000 times.

Each run consists of the three phases preparation, execution and clean-up. During preparation, the whole environment of the experiment process must be reset to a defined initial working condition. Software agents are initialized. In testbeds, for example, network packets generated in previous runs must be dropped on all participants. Preliminary measurements can be done which are needed for compensation of incomplete control over the environment, such as clock offsets for all participants. During execution, the actual experiment process is executed, observed and recorded. Clean-up takes care of correctly terminating a run on each participant. All steps will be repeated during each run, this has to be considered when estimating the total time an experiment needs to finish.

2) Description of Processes: ExCovery provides common mechanisms to control execution of the defined processes. Two types of processes can be differentiated, depending whether they relate to abstract nodes or to the environment. Abstract node processes are mapped to real nodes during experiment execution, such as protocol actions or fault injection processes. Environment processes are performed by all nodes, such as dropping packets on all network interfaces to reset the environment. Every process is described as a sequence of actions. Processes run concurrently on the nodes so to specify this sequence, one needs to consider timing and desired or necessary dependencies. *ExCovery* defines methods for synchronization of the execution to provide basic flow control.

wait_for_time Lets the process wait for a fixed delay in seconds.

wait_for_event Lets the process wait until the specified event is registered on any participant. An event can be specified by its name, location and any of its parameters. The location is either a single abstract node or a subset of nodes specified by an actor role. Event parameters can be of diverse types. If omitted, they default to "any". A time-out in seconds can be set.

wait_marker Creates a time stamp that will be used by the next `wait_for_event` call, which considers only events occurring after that time stamp.

event_flag Used to create local events to let process actions depend directly on each other.

Besides these flow control functions, there are process specific actions, environment actions and manipulation actions. Each action can have a list of parameters. This allows to describe manifold scenarios. In Section V, service discovery as an experiment process is described to illustrate this. Manipulation Processes are described in Section IV-D3. Figure 6 shows a code fragment where the different processes are defined, without the actual sequences of actions that will be described later. Among the node processes, the role $actor0$ is defined and as possible actor nodes, the abstract nodes $fact\_nodes$ from the factor list are referenced. Environment processes do not need a definition of nodes.

D. Fault Injection and Environment Manipulation

ExCovery has a concept for intentional manipulations done on participant nodes and on their network environment. Manipulations cover direct fault injections cause failures in a targeted area. Fault provocation is used when direct injection is not desirable or possible and characterizes actions that are known to provoke failures in the targeted area. The main faults considered are communication faults. *ExCovery* provides a simplified fault model to allow for the description of basic fault behavior. Fault injection processes can have common

```
<processes max_run_time="120">
  <node_processes name="...">
    <process_parameters>
      <actor_node_map><factorref id="fact_nodes"/>
      </actor_node_map>
    </process_parameters>
    <actor id="actor0" name="SM">
      <!-- list of node process actions -->
    </actor>
  </node_processes>
  <env_process>
    <!-- list of environment process actions -->
  </env_process>
</processes>
```

Fig. 6. Template for the description of node and environment processes.

parameters describing their temporal behavior: *duration*, *rate* and *randomseed*. The *duration* specifies the amount of time a fault should be applied to the target. The *rate* specifies a percentage of a given duration in which a fault is active. The fault is active in one continuous block, its activation time is chosen randomly using the *randomseed*.

1) Fault Injections: In addition to the common fault parameters, injections can have individual parameters to further define their behavior. The mechanisms for communication fault injection are explained in the following. Whenever the term packet is used, it refers to packets belonging to the experiment process. It should be noted that all injected faults add up to already existing communication faults in the target platform.

Interface fault No messages are transmitted or received on the specified interface in the specified direction as long as this fault is active. Direction can be receive, transmit, both, or chosen randomly.

Message loss Defines a given probability for every packet of the experiment process to be dropped. Direction is analogous to the interface fault.

Message delay Applies a given constant delay to every packet.

Path loss and path delay Path loss and delay are message loss and delay faults, selectively affecting only the communication between the target and a given second node.

2) Environment Manipulations: Environment manipulations are applied on a global level and involve more than one node, possibly all specified environment nodes. Manipulations include the previously defined fault injections. Additionally, the following manipulations can be applied to create different conditions for the experiment.

Traffic generator Creates network load between a given number of node pairs. Each pair bidirectionally communicates at a given data rate (see also Figure 5). Pairs can be randomly chosen from the acting nodes, non-acting nodes or all nodes. They vary from run to run as determined by a switch amount parameter.

Drop all packets All experiment nodes stop receiving, sending and forwarding the experiment process packets.

Every fault injection and environment manipulation but the traffic generator is started only once and without a given duration, needs to be explicitly stopped. Given is just the default list supported by *ExCovery*. There also is a generic function, which has an arbitrary list of parameters that are given to the acting nodes to be executed. However, an experimenter should

```
<env_process>
  <env_actions>
    <event_flag><value>"ready_to_init"</value></event_flag>
    <env_traffic_start>
      <bw><factorref id="fact_bw" /></bw>
      <choice>0</choice>
      <!-- this causes identical randomization in
          ↪ replications -->
      <random_switch_amount>"1"</random_switch_amount>
      <random_switch_seed>
        <factorref id="fact_replication_id" />
      </random_switch_seed>
      <random_pairs><factorref id="fact_pairs" />
        </random_pairs>
      <random_seed><factorref id="fact_pairs"/>
        </random_seed>
    </env_traffic_start>
    <wait_for_event>
      <event_dependency>"done"</event_dependency>
    </wait_for_event>
    <env_traffic_stop />
  </env_actions>
</env_process>
```

Fig. 7. Illustrative example of environment process for traffic generation.

preferably extend *ExCovery* by defining a plugin with new functions and their implementation.

3) Description of Manipulation Processes: Manipulation and fault injection processes are defined in the experiment description as a series of actions an events. This list is executed in sequential manner and can contain flow control functions as described in Section IV-C2. A node manipulation process is created for each abstract node it is specified for while the environment manipulation process is implicitely supported on all nodes. The specific actions activate or deactivate the faults and manipulations as detailed in Section IV-D1 and IV-D2. One event is generated by each action to signal its start or stop, respectively. Parameters of these actions can be constant or varied during experiment execution. Variation is realized by references to factors instead of fixed values. The manipulation actions can be used to extend the experiment process description. An experimenter can place desired functions in line with the other functions used in the experiment process, or into separate manipulation processes that run along with each experiment process. This depends on whether the faults and manipulations shall be synchronous with the experiment process or autonomous. Figure 7 shows a shortened listing of a traffic generation process. After generating a *ready_to_init* event, it uses the factors from Figure 5 to choose and configure traffic generation by a set of environment nodes, switching one pair of nodes in every run. The manipulation remains active until an event *done* is registered.

E. Description of Specifics

To enable *ExCovery* to instantiate an abstract experiment description on a concrete platform, some specific settings are necessary. For reasons of brevity, not all settings will be mentioned in this paper. This section explains the most important of these settings, which are included in the experiment description. For execution on a specific platform, a mapping of abstract and environment nodes to concrete usable nodes of the platform is required. This mapping can change from one experiment to another on the same platform due to the availability of nodes or when deliberately changing the nodes. *ExCovery* identifies nodes by their host name and IP address. The host name should be constant during an experiment run.

```
<platform_specs>
  <spec_node_mapping>
    <spec_actor_map abstract_id="A" id="t9-105"
        ip="172.18.17.22" />
    <spec_actor_map abstract_id="B" id="a3-119"
        ip="172.18.17.173" />
    <spec_env_map id="a3-005" ip="172.18.17.178" />
    <spec_env_map id="a3-010" ip="172.18.17.180" />
    <spec_env_map id="t9-k61" ip="172.18.17.92" />
    <spec_env_map id="t9-k63" ip="172.18.17.46" />
  </spec_node_mapping>
</platform_specs>
```

Fig. 8. Platform specification in the experiment description.

When an IP addresse changes due to reconfiguration of a network interface, for example after a injection of such a fault, an event is generated to signal this. Finally, an experimenter can define a list of special parameters in the description file that can be used within the experimentation environment to expose specific parameters used in the implementation to the description file. This allows platform specific modifications on the *ExCovery*'s execution program to be reused for many experiments without having to modify the implementation each time. Figure 8 illustrates a compact version of a platform specification. Two actor nodes and four environment nodes exist. Actor nodes map to an abstract node id that has been previously defined. All nodes have a unique identifier and a network address that can later be used to analyze the recorded event and packet lists.

F. Measurement Storage and Conditioning

ExCovery provides four levels of storage for experiments, with defined data structures. This allows reusable data access functions among experiments. The first storage level is the abstract experiment description itself, stored in an XML document. This document can be exchanged and loaded for execution and analysis. The second level is the intermediate storage for all concrete experiment data: experiment results and the software artifacts used during execution. Each log file and measurement is stored corresponding to a run identifier and associated to the node it originates from. Currently, *ExCovery* uses a special hierarchy on a file system to store second level data. On the way to the third storage level, data are conditioned by first evaluating the synchronization measurements taken during the experiment (see Section IV-B3) and unifying the time base of all second level measurements. Then, the event list and captured packets are split up into single entries. Data from the second level plus the experiment description are then stored into a single package on the third level. This package represents one complete experiment and is preferably stored as a database to unifiy and accelerate data access and extraction methods. Facilitating exchange of experiments, *ExCovery* currently stores the third level in a file based relational SQLite database. The fourth level describes the integration of multiple experiments into a single repository to facilitate comparison and analysis covering multiple experiments. To date, *ExCovery* does not realize this level.

Table I shows a representation of a subset of the tables and their attributes on the third level. The table `ExperimentInfo` represents the experiment as a whole and contains only one tuple made of the abstract experiment description, the version of *ExCovery* and a descriptive name and comment. `Logs` contains all raw log files and `EEFiles`

TABLE I. TABLES AND ATTRIBUTES OF CURRENT STORAGE CONCEPT

Table	Attributes
ExperimentInfo	ExpXML, EEVersion, Name, Comment
Logs	NodeID, Log
EEFiles	ID, File
ExperimentMeasurements	ID, NodeID, Name, Content
RunInfos	RunID, NodeID, StartTime, TimeDiff
ExtraRunMeasurements	RunID, NodeID, Name, Content
Events	RunID, NodeID, CommonTime, EventType, Parameter
Packets	RunID, NodeID, CommonTime, SrcNodeID, Data

the used *ExCovery* executables to support transparency and assist development. In `ExperimentMeasurements`, specific named measurements are stored that are done once per experiment. As for run based data, `RunInfos` contains for each run and node the start time of the run and the offset of the node clock to the reference clock. Custom measurements are stored in `ExtraRunMeasurements`. The table `Packets` contains for each packet the common time stamp of detection, its originating node and the raw packet data. The `Events` table lists all recorded events and their parameters, identified by the run, the originating node and a common time stamp. This schema represents a preliminary approach to store data. Several future improvements are possible, for example by using a dimensional database model to store experiments in a data warehouse structure.

V. ABSTRACT SERVICE DISCOVERY PROCESSES DESCRIPTION

To demonstrate the transfer of the described concepts to concrete distributed processes that are to be examined in experiments, we will now explicate how to describe generic service discovery (SD) as an experiment process to be used within *ExCovery*. It provides a temporal and causal sequence of all actions of the participating nodes as introduced in Section III, facilitating flow control functions from Section IV-C2. The description can contain multiple actors representing SMs, SUs, or SCMs. For each actor a number of instances can be created to represent all participants of the SD process. The model developed in [15] defines a set of actions for a generic SD process, namely “Configuration Discovery and Monitoring”, “Registrations and Extension”, “Service-Description Discovery and Monitoring”, and “Variable Discovery and Monitoring”. Only these main actions are considered in the SD process description, with an optional list of parameters to specify concrete variants of the actions. The description does not intend to model an SD protocol (SDP) specific behavior in detail, but to give an abstract description of a service discovery scenario. The details of executing the description are left to the SDP implementation, so that multiple implementations which adhere to the same SD concepts can be compared in experiments. However, executing SDPs are allowed to generate user specified events which will be recorded by *ExCovery*. Actions that can be executed on participating SD nodes are described as follows.

Init SD Mandatory action to allow participation of a node in the SD. Represents “Configuration Discovery and Monitoring”. Depending on the SDP, discoverable items such and scopes and SCMs are discovered and each node’s unique identity is established. This action reads as parameter the role as either SCM or one of SU and SM. An optional list of parameters configures user specified parameters of the used SDP. When the SCM parameter is used, the node generates a `scm_started` event. If an SM registers its service on an SCM node, a `scm_registration_add` event is generated with the registering node’s identification as parameter. Analogously, when a registration is revoked or changed, the respective events `scm_registration_del` and `scm_registration_upd` are generated. In a hybrid architecture, SU and SM agents keeps looking for SCMs and emit `scm_found` events when a SCM has been discovered. When initialization is complete, `sd_init_done` is emitted.

Exit SD Stops the previously started role and all assigned searches and publishings, emitting `sd_exit_done` upon completion. To participate again in the SD process, a node needs to re-run init.

Start searching On SU and SM nodes initiates a continuous SD process for a given service type, generating the event `sd_start_search`. Refers to the group of “Service-Description Discovery and Monitoring” functions. *ExCovery* does not distinguish among *passive*, *aggressive*, or *directed discovery* (SCM). A service is considered discovered during search when its complete description has been received, when the event `sd_service_add` will be emitted with the found service’s identifier as parameter. Analogously, when a service is becomes unavailable, the event `sd_service_del` is generated

Stop searching A previously started search is stopped. Includes removal of any notification request previously given to SCMs. Event `sd_stop_search` is generated at the time the search is stopped.

Start publishing Starts publishing an instance of a given service type, generating a `sd_start_publish` event. Refers to the group of “Registrations and Extension” functions, such as registration on an SCM and management of registrations.

Stop publishing Gracefully stops publishing of a given service type. Includes further actions like aggressively sending revocation messages or de-registration on SCMs. Generates a `sd_stop_publish` event upon completion.

Update publication Updates a previously published service description upon change. Covers any underlying functions related to registration on SCMs. Generates an event `sd_service_upd` with the service identifier as parameter before the update is executed.

Figures 9 and 10 show descriptions of two-party SD processes for SU and SM roles using the introduced actions and events. The SM role in Figure 9 basically starts publishing and and continues until a *done* event is registered. The SU role in Figure 10 is considerably more complex. An SU waits first for all SMs to emit their $sd_start_publish$ event, then for the environment to register the $ready_to_init$ event. It will then start searching and finish either when all SMs have been discovered, having generated their respective $sd_service_add$ events or when the deadline of 30 seconds has been reached. In either case, *done* is generated and the clean-up phase begins. An example SD scenario is depicted in Figure 11. It shows a single active SD in a two-party architecture with a timeline for each actor SU and SM. Actions are shown as white circles, events as black circles. Where events are not labeled they inherit the

```
<actor id="actor0" name="SM">
  <sd_actions>
    <sd_init />
    <sd_start_publish />
    <wait_for_event>
      <event_dependency>"done"</event_dependency>
    </wait_for_event>
    <sd_stop_publish />
    <sd_exit />
  </sd_actions>
</actor>
```

Fig. 9. SD process in a two-party architecture. Publisher role.

```
<actor id="actor1" name="SU">
  <sd_actions>
    <wait_for_event>
      <from_dependency>
        <node actor="actor0" instance="all"/>
      </from_dependency>
      <event_dependency>"sd_start_publish"
        </event_dependency>
    </wait_for_event>
    <wait_for_event>
      <event_dependency>"ready_to_init"
        </event_dependency>
    </wait_for_event>
    <sd_init />
    <wait_marker />
    <sd_start_search />
    <wait_for_event>
      <from_dependency><node actor="actor1" instance="all"/>
      </from_dependency>
      <event_dependency>"sd_service_add"</event_dependency>
      <param_dependency><node actor="actor0" instance="all"/>
      </param_dependency>
      <timeout>"30"</timeout>
    </wait_for_event>
    <event_flag><value>"done"</value></event_flag>
    <sd_stop_search />
    <sd_exit />
  </sd_actions>
</actor>
```

Fig. 10. SD processes in a two-party architecture. Requester role.

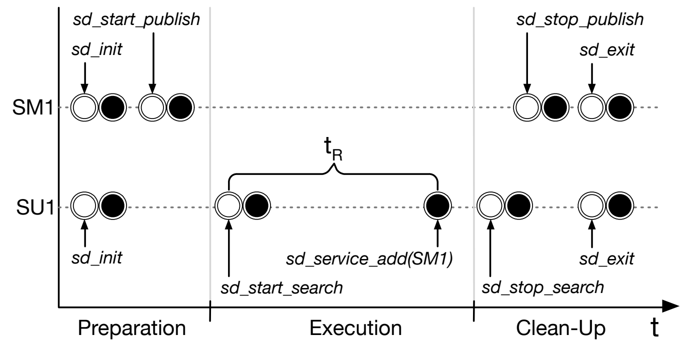

Fig. 11. Visualization of a one-shot discovery process.

label of the preceding action. In the preparation phase, SU and SM initialize themselves. This phase ends a fixed time after the event `sd_start_publish` from SM1 is registered, to let unsolicited announcements of SM1 pass. SU1 then starts a search, beginning the execution phase. After a time t_R the service is discovered and an event `sd_service_add` is generated on SU1. The SD scenario finishes here, in the clean-up phase searches and publications are stopped and the SD system shut down.

VI. Prototype Implementation

An *ExCovery* prototype has been created with the aim of being reusable on diverse platforms. It abstracts the handling of the XML experiment description and the resulting run sequence and parameter variations in separate classes that can be instantiated by programs to analyze, visualize, trace or export experiment related data. The core of *ExCovery* is implemented using the the Python programming language. As the first supported platform, an implementation for the wireless DES testbed at Freie Universitt Berlin (FUB) [22] is provided. To demonstrate feasability, an implementation for the SD process in Section V exists. This section gives a quick overview of the prototype, for a comprehensive description of the implentation the reader should refer to [17]. It should be noted that neither code listing presented in this paper is complete and has been shortened for illustrative purposes. The full code and descriptions are available on request and a repository will shortly be publicly accessible.

A. Software components

In accordance to the developed concept the prototype is composed of one controlling entity (master) and a set of controlled entities (nodes) as depicted in Figures 3 and 12. Master and nodes are connected in a centralized client-server architecture with a dedicated communication channel. They communicate synchronously using extensible markup language remote procedure calls (XML-RPC) [23].

The controlling *ExperiMaster* maintains a list of objects corresponding to the active nodes in the experiment, on which actions will be executed. A node object presents the functions of one node to the master program via XMLRPC and uses locking to allow only one access at a time. Which action is executed at which time is specified in process descriptions loaded from the experiment description file. The master creates an experiment process thread and a fault thread for each abstract node in the description. A single thread is created for the environment manipulations. The actions performed by this thread and the management actions performed by the main program can be executed concurrently on all nodes. The *NodeManager* is the central component of the nodes participating in experiments. It handles remote procedure calls (RPCs) coming from *ExperiMaster*. Basic procedures exposed via RPC are the actions for management, fault injection, environment manipulation and the experiment process actions as defined in Sections IV and V. The implementation of these functions can be delegated to sub-components. For example, the experiment process actions in the context of this work refer to SD actions that are implemented by the avahi software package. Components on a node use the event generator to signal the occurrence of events, as defined in Section IV-B1.

To allow analysis of properties outside the scope of the *ExCovery* processes, for example packet loss and delay, a network packet tagger is provided. It remains running in the background on each node. The tagger adds an option to the header of each selected IP packet and writes a 16 bit identifier to it, incrementing the identifier with each packet. Additionally, *ExCovery* includes a set of Python scripts to collect, condition and store experiment results in a database.

The presented concept and implementation generally supports multiple SDPs. They need to provide a Linux implementation which provides an interface to fundamental SDP

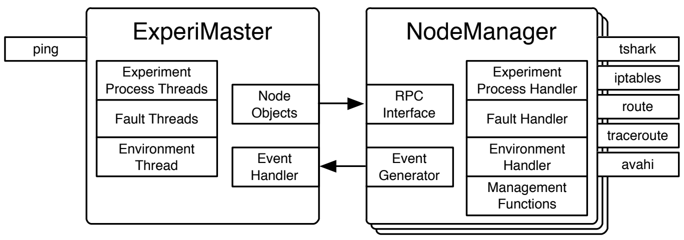

Fig. 12. Execution components of the provided implementation.

operations, as represented by the actions in Section V. For the prototype, the Zeroconf SDP suite Avahi [24] was used and modified to allow the association of request and response pairs. This allows analysis of response times not only on SD operation level (t_R in Figure 11) but on the level of individual SD request and response packets, which by default is not the supported in Zeroconf SDPs. A set of functions exist for extraction and analysis of event and packet based metrics. As a time-critical operation, one key property of SD is *responsiveness* – the probability that a number of SMs is found within a deadline, as required by the application calling SD. *ExCovery* was originally developed to support and validate research on SD responsiveness, as in [25], [26]. Due to space constraints, this work covers the abstract description of these experiments, their results will be published in future work.

VII. Conclusion and Outlook

This work presents the experimentation framework *ExCovery*, to support experiments on the dependability of distributed processes. It provides concepts that cover the description, execution, measurement and storage of experiments, to foster their transparency and repeatability. The description covers the specification of the individual processes of an experiment and their actors: Fault injection, environment manipulation and the main process under experimentation are expressed as interdependent series of actions and events. Execution takes care of controlling the individual nodes during experiment runtime, to make sure each run of an experiment has a clean and defined environment and each node acts according to the experiment description. *ExCovery* manages series of experiments and recovers from failures by resuming aborted runs. Measurements are taken both on the level of process actions and events and on the level network packets. They are stored in a unified database format to faciliate sharing and comparison of results. As a case study, we provided an abstract description of service discovery (SD) as experiment process. *ExCovery* has been tried and refined in a manifold of SD dependability experiments over the last two years. A working prototype runs on the wireless DES testbed at Freie Universität Berlin.

References

[1] H.-J. Rheinberger, *Experiment, Differenz, Schrift: zur Geschichte epistemischer Dinge*. Basilisken-Presse, Verlag Natur und Text, 1992.

[2] A. Dean and D. Voss, Eds., *Design and Analysis of Experiments*, ser. Springer Texts in Statistics. Springer, 1999.

[3] D. C. Montgomery, *Design and Analysis of Experiments*, 7^{th} ed. John Wiley and Sons, Inc., 2009.

[4] R. A. Bailey, *Design of Comparative Experiments*, ser. Cambridge Series in Statistical and Probabilistic Mathematics. Cambridge University Press, April 2008, vol. 25.

[5] K. Hinkelmann and O. Kempthorne, *Design and Analysis of Experiments, Introduction to Experimental Design*, 2^{nd} ed. John Wiley and Sons, Inc., January 2008, vol. 1.

[6] W. F. Tichy, "Should computer scientists experiment more?" *Computer*, vol. 31, no. 5, pp. 32–40, May 1998.

[7] D. G. Feitelson, "Experimental computer science: The need for a cultural change," December 2006, manuscript.

[8] B. Milic and M. Malek, "Properties of wireless multihop networks in theory and practice," in *Guide to Wireless Ad Hoc Networks*, ser. Computer Communications and Networks, S. Misra, I. Woungang, and S. Chandra Misra, Eds. Springer, 2009, pp. 1–26.

[9] A. Quereilhac, M. Lacage, C. Freire, T. Turletti, and W. Dabbous, "NEPI: An integration framework for network experimentation," in *19th International Conference on Software, Telecommunications and Computer Networks (SoftCOM)*. IEEE, September 2011, pp. 1–5.

[10] Y. Wang, "Automating experimentation with distributed systems using generative techniques," Ph.D. dissertation, University of Colorado, Boulder, CO, USA, 2006.

[11] Y. Wang, A. Carzaniga, and A. L. Wolf, "Four enhancements to automated distributed system experimentation methods," in *30th International Conference on Software Engineering (ICSE)*. ACM, 2008, pp. 491–500.

[12] T. Erl, *Service-Oriented Architecture: Concepts, Technology, and Design*, 1^{st} ed., ser. The Prentice Hall Service Technology Series from Thomas Erl. Prentice Hall PTR, August 2005.

[13] C. A. Flores-Cortés, G. S. Blair, and P. Grace, "An adaptive middleware to overcome service discovery heterogeneity in mobile ad hoc environments," *IEEE Distributed Systems Online*, vol. 8, no. 7, July 2007.

[14] S. Brüning, S. Weissleder, and M. Malek, "A fault taxonomy for service-oriented architecture," in *Informatik-Berichte*. Humboldt-Universität zu Berlin, 2007, vol. 215.

[15] C. E. Dabrowski, K. L. Mills, and S. Quirolgico, "A model-based analysis of first-generation service discovery systems," NIST National Institute of Standards and Technology, Gaithersburg, MD, USA, Tech. Rep. ADA523379, October 2005.

[16] G. G. Richard, "Service advertisement and discovery: Enabling universal device cooperation," *IEEE Internet Computing*, vol. 4, no. 5, pp. 18–26, Sep-Oct 2000.

[17] S. Wanja, "Experimentation environment for service discovery," Master's thesis, Humboldt-Universität zu Berlin, February 2012.

[18] M. S. Thompson, "Service discovery in pervasive computing environments," Ph.D. dissertation, Virginia Polytechnic Institute and State University, Blacksburg, Virginia, USA, July 2006.

[19] F. Zhu, M. W. Mutka, and L. M. Ni, "Service discovery in pervasive computing environments," *IEEE Pervasive Computing*, vol. 4, pp. 81–90, 2005.

[20] W. K. Edwards, "Discovery systems in ubiquitous computing," *IEEE Pervasive Computing*, vol. 5, no. 2, pp. 70–77, 2006.

[21] T. Zseby, C. Henke, and M. Kleis, "Packet tracking in planetlab europe – a use case," in *Testbeds and Research Infrastructures. Development of Networks and Communities*, ser. Lecture Notes of the Institute for Computer Sciences, Social Informatics and Telecommunications Engineering, T. Magedanz, A. Gavras, N. Thanh, and J. Chase, Eds. Springer, 2011, vol. 46, pp. 265–274.

[22] B. Blywis, M. Günes, F. Juraschek, and O. Hahm, "Properties and topology of the DES-testbed," Freie Universität Berlin, Tech. Rep. TR-B-11-02, March 2011.

[23] D. Winer, "Extensible markup language remote procedure calls," Specification, June 2003. [Online]. Available: http://xmlrpc.scripting.com/spec.html

[24] "The avahi project," February 2014. [Online]. Available: http://avahi.org

[25] A. Dittrich and F. Salfner, "Experimental responsiveness evaluation of decentralized service discovery," in *27th International Parallel & Distributed Processing Symposium, Workshops and Phd Forum (IPDPSW)*. Boston, MA, USA: IEEE, May 2013, pp. 1–7.

[26] A. Dittrich, B. Lichtblau, R. Rezende, and M. Malek, "Modeling responsiveness of decentralized service discovery in wireless mesh networks," in *MMB & DFT*, ser. Lecture Notes in Computer Science, K. Fischbach and U. R. Krieger, Eds. Springer, March 2014, vol. 8376, ch. 7, pp. 88–102.