ETH
Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

Prof. Dr. L. Thiele
Frühlingssemester 2010

SEMESTERARBEIT

Management Software für ein Wireless Sensor Netzwerk Testbed

Autor:
Stefan Kronig

Betreuer:
Matthias Woehrle
Christoph Walser

Inhaltsverzeichnis

Kapitel 1

Einleitung

Drahtlose Sensornetzwerke (wireless sensor networks, WSN) werden verwendet, um Daten an verschiedenen geographischen Orten zu sammeln und anschliessend an eine zentrale Station zu übermitteln (z. B. Überwachungssysteme, Umweltdaten, Aufenthaltsort von Tieren einer Herde, usw.). Ein WSN besteht aus einzelnen drahtlosen Sensoren, sogenannten Sensorknoten. Diese führen zum einen die eigentlichen Messungen durch, zum Anderen kommunizieren sie aber auch mit ihren Nachbar-Knoten und bilden so ein Netzwerk, durch welches die gesammelten Daten an einen zentralen Knoten übermittelt werden können. Da WSNs häufig an schwer zugänglichen Orten installiert sind, ist es wichtig, dass sie bereits beim Installieren zuverlässig funktionieren und nicht nachträglich noch Updates benötigen. Dedizierte Test-Plattformen (Testbeds) haben sich als sehr nützliches Werkzeug erwiesen, um WSNs bereits im Labor ausgiebig und realitätsnah testen zu können. Ein Testbed bildet ebenfalls ein Netzwerk, mittels dem die Sensorknoten überwacht, ihnen Befehle erteilt oder sie neu programmiert werden können.

FlockLab [1] ist ein Testbed, das am Institut für Technische Informatik und Kommunikationsnetze (TIK) der ETH Zürich entwickelt wurde. Beim FlockLab ist jeder Sensorknoten an einen sogenannten Observer angeschlossen. Die Observer bestehen aus einem embedded PC sowie der benötigten Peripherie, um Messungen am Sensorknoten durchzuführen, ihm Befehle zu erteilen usw. [2]. Das Spezielle an FlockLab gegenüber anderen Test-Plattformen ist, dass 1. auf jedem Knoten eine Leistungsmessung zur Verfügung steht, und 2. die Zeit auf den Observern sehr genau synchronisiert ist (auf einige μs genau). Letzteres ist notwendig, um Vorgänge, welche im Netzwerk zu einem bestimmten Zeitpunkt eingetreten sind, auch als zeitsynchron registrieren zu können.

Das Herzstück der Observer ist ein Gumstix[1] embedded PC, auf dem Linux läuft. Um Messungen durchzuführen, stehen dem Benutzer verschiedene Services zur Verfügung. Diese laufen als Daemons (Hintergrundprozesse) auf dem Gumstix und können nach Bedarf gestartet, konfiguriert und gestoppt werden (Details dazu in [1]). Um einen Test auf dem Testbed auszuführen, musste man jeweils Testbed re-

[1] Gumstix inc., gumstix - dream, design, deliver. `http://www.gumstix.com/`

servieren und sich zur reservierten Remote auf den Observern anmelden, um jedem Observer einzeln die Befehle zu erteilen, damit er die Software auf den Sensorknoten überträgt und zur richtigen Zeit die Messungen startet. Dies ist sehr mühsam und nicht praktikabel, wenn das Testbed unter mehreren Benutzern aufgeteilt werden soll.

In Zukunft sollen die Benutzer ihre Tests im Voraus registrieren können (voraussichtlich mittels eines Web-Interfaces). Mittels dieses Interfaces wird es auch möglich sein, die Konfigurationsdaten und die Images für die Sensorknoten auf einen FlockLab-Server zu laden. Die hochgeladenen Testdaten werden, sofern sie gültig und fehlerfrei sind, in die Datenbank eingefügt. Ein Scheduler erstellt dann automatisch einen Ablaufplan und führt zur geplanten Zeit den Dispatcher aus. Dieser verteilt den Test auf die Observer, bereitet die Services vor und startet den Test. Der gesamte Ablauf, wie ein Test registriert, geplant und ausgeführt werden soll, ist in Abbildung 1.1 dargestellt. Das Ziel dieser Semesterarbeit war es, den Scheduler und den Dispatcher zu entwerfen. Die Überlegungen zum Scheduler werden in Kapitel 4 behandelt. Die Überlegungen zum detaillierten Ablauf sowie zur Implementation des Dispatchers befinden sich in Kapitel 3 dieser Semesterarbeit.

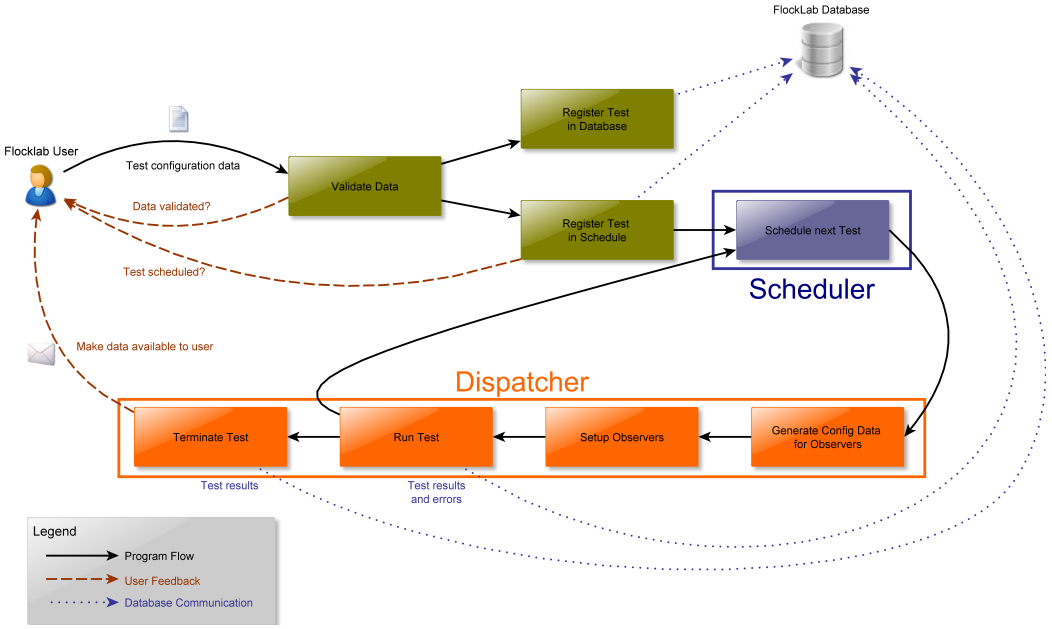


Abbildung 1.1: Überblick, wie künftig ein Test auf dem FlockLab registriert, geplant und ausgeführt werden soll. In den Kästchen sind die Teile, welche in dieser Semesterarbeit behandelt werden.

Kapitel 2

Verwandte Arbeiten

Im folgenden Kapitel möchte ich einige dem FlockLab verwandte Arbeiten vorstellen. Es sind dies andere Testbeds sowie Methoden, um ein Testbed unter mehreren Benutzern zu teilen.

2.1 Beispiele von WSN Testbeds, Vergleich zum FlockLab

Das *Deploymend Support Network (DSN)* [3] stellt einen möglichen Ansatz dar, um ein WSN zu überwachen. Dabei ist das DSN selbst wiederum ein WSN, durch welches ein anderes WSN überwacht wird. DSN wurde ebenfalls am TIK entwickelt. Das zu überwachende Sensornetzwerk besteht aus TinyNodes[1], das Support-Netzwerk aus BTnodes[2]. Beim FlockLab wurde diese Idee weiter entwickelt, das "Support-Netzwerk" ist nun aber kein WSN mehr, sondern besteht aus den sogenannten Observern, welche via (W-)LAN zu einem FlockLab-Server verbunden sind. Dies bringt Vorteile bezüglich dem Datendurchsatz. Zudem haben die Observer spezielle Hardware, um den Leistungsverbrauch der Sensorknoten zu messen, was mit den BTnodes nicht möglich ist (nur mit extern angeschlossener Hardware).

MoteLab [4] wurde in Harvard entwickelt, hat LAN als Backbone, hat aber nur an einem Node eine Strommessung. Zudem ist es um einiges grösser (184 Nodes im Juni 2010) als das FlockLab. Die Autoren von MoteLab behaupten von sich, dass sie ihr Web-Interface sehr benutzerfreundlich gestaltet haben. Vielleicht wird man hier also Ideen übernehmen können für das geplante Web-Interface des FlockLabs. Dieses ist aber nicht Teil meiner Semesterarbeit. Speziell für das MoteLab ist, dass es einen direkten Zugang auf die Sensorknoten bietet (direct node access), indem die serielle Schnittstelle via TCP/IP weitergeleitet wird. So kann der Benutzer einen eigenen Server implementieren und dem WSN auch Live-Befehle erteilen. Beim FlockLab wurde ein anderer Ansatz gewählt, nämlich dass ein Test vollkommen

[1] Shockfish SA. TinyNode. `http://www.tinynode.com/`

[2] BTnote Project @ ETH Zürich. BTnodes - A Distributed Environment for Prototyping Ad Hoc Networks. `http://www.btnode.ethz.ch`

autonom ablaufen soll. Dazu gibt es einen Service, welcher die Daten der seriellen Schnittstelle lesen kann. Zudem ist ein Service geplant, welcher zu einer bestimmten Zeit Daten an die Schnittstelle senden kann.

Das MoteLab ist ähnlich aufgebaut, wie das FlockLab, FlockLab stellt jedoch eine neuartige, service-orientierte Testumgebung zur Verfügung. Das heisst, es stellt Dienste für verschiedenartige Messungen zur Verfügung und nicht nur z. B. den Zugriff auf die serielle Schnittstelle der Sensorknoten. Dieser neuartige Ansatz benötigt in den meisten Fällen auch eine neue Software, es ist daher nur selten möglich, Software von anderen Projekten zu übernehmen.

2.2 Testbed unter mehreren Benutzern teilen

Ein eigenes Testbed stellt für eine Forschungsanstalt eine gewisse Investition und auch Aufwand dar, es macht also Sinn, dieses unter mehreren Benutzern aufzuteilen. Wie auch bei anderen geteilten Ressourcen, soll diese Aufteilung möglichst fair erfolgen. Es soll beispielsweise nicht möglich sein, dass ein Benutzer das Testbed während längerer Zeit für sich beanspruchen kann, obwohl andere es auch benutzen möchten. Dazu wurden verschiedene Methoden vorgestellt. Beim MoteLab ist beispielsweise die Zeit beschränkt, wie lange ein Benutzer das Testbed im Voraus reservieren kann. Es gibt aber keine Beschränkung, wie lange man das Testbed total benutzen darf.

Mirage [5] stellt eine Methode vor, um ein Testbed mehreren Benutzern zugänglich zu machen, welche von der Finanzwelt inspiriert wurde: Die Ressourcen werden in regelmässigen Zeitabständen mittels kombinatorischer Auktion “versteigert”. Die Benutzer können ihre Wünsche für Ressourcen (z. B. 32 Sensorknoten, 8 Stunden lang, irgendwann während den nächsten drei Tagen) zusammen mit einem Gebot abgeben. Basierend auf Angebot und Nachfrage werden dann die Jobs auf die einzelnen Sensorknoten verteilt, so dass das Testbed möglichst viel Geld einbringt. Da Benutzer dort hoch bieten, wo sie für sich den grössten Nutzen erhoffen, wird so automatisch der grösste, totale Nutzen für die Benutzer erzielt. Ein Resource Discovery Service stellt fest, welche Sensorknoten frei sind und was für Ressourcen (z. B. Sensoren, Frequenzband, ...) sie haben.

Beim FlockLab ist es die Aufgabe des Schedulers, aufzuteilen, welcher Test wann auf welchem Sensorknoten ausgeführt wird. Der Scheduler soll auch sicherstellen, dass diese Aufteilung fair abläuft. Einschränkungen für Benutzer (Quotas, Versteigerung oder ähnlich) sind momentan allerdings keine geplant, da nicht mit einer allzu hohen Auslastung gerechnet wird. Weitere Überlegungen zum Scheduler sind im Kapitel 4 dieser Arbeit zu finden.

Kapitel 3

Dispatcher

In diesem Kapitel wird der Dispatcher vorgestellt. Seine Aufgabe ist es, einen bestimmten Test auf die Observer zu verteilen und auszuführen.

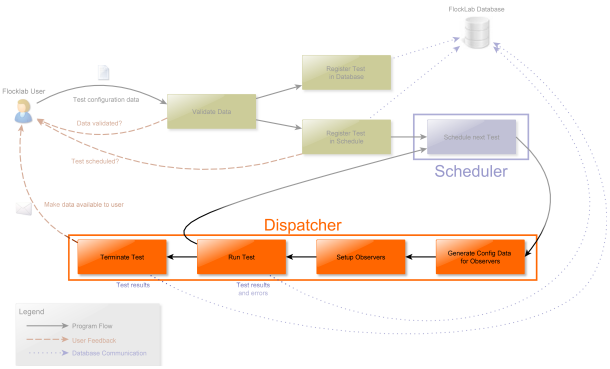


Abbildung 3.1: Der Dispatcher im gesamten Testablauf.

3.1 Ablauf

Der Dispatcher (Deutsch etwa: Abfertiger, Disponent) ist derjenige Teil der Software, welcher die Aufgabe hat, einen Test auf den Observern auszuführen. Seine Funktionsweise ist in Abbildung 3.2 zusammengefasst, ein detaillierter Ablaufplan ist im Anhang A zu finden.

Da der Dispatcher die Observer koordinieren soll, läuft er auf einem FlockLab-Server. Er sendet Kommandos an die Observer, welche diese ausführen und mit einer Erfolgs- oder Fehlermeldung quittieren müssen. Die genaue Arbeitsaufteilung zwischen Server und den Observern ist im detaillierten Ablaufplan ebenfalls ersichtlich. Er wird vom Scheduler rechtzeitig vor Beginn des Tests aufgerufen. Dieser übergibt ihm als Argument die ID des auszuführenden Tests.

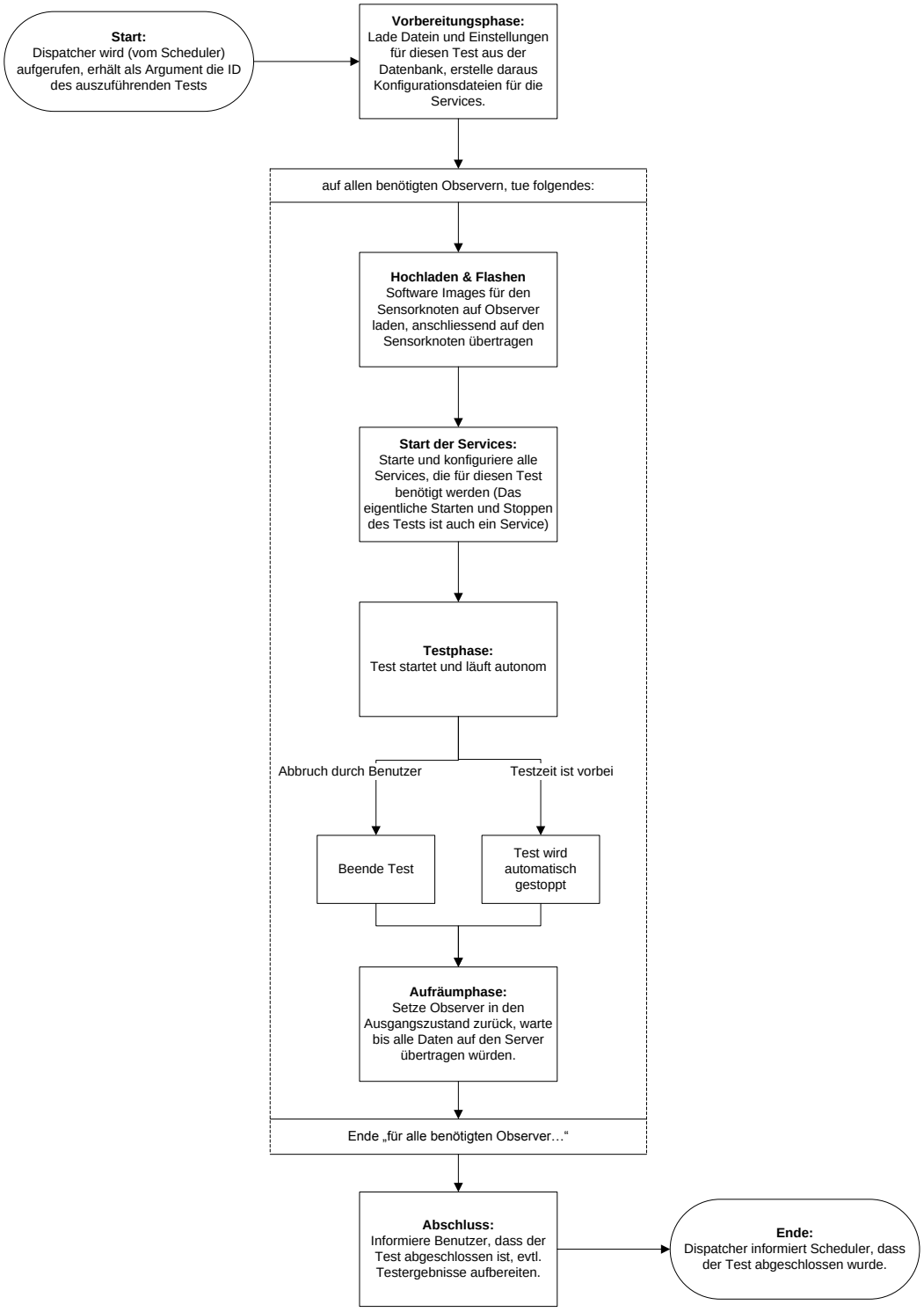


Abbildung 3.2: Ablaufplan des Dispatchers, zusammengefasst.

3.1.1 Vorbereitungsphase

Auf dem FlockLab-Server sind alle Informationen über vergangene, laufende und zukünftige Tests in einer Datenbank gespeichert. Der Dispatcher muss also als erstes alle Informationen aus der Datenbank holen, die er für den aktuellen Test benötigt. Dies sind u. a. eine Liste der benötigten Observer und Services, die Software-Images für die Sensorknoten sowie die Konfigurationen der Services. Letztere müssen noch verarbeitet werden, da auf den Observern ein anderes Format für Konfigurationsdateien verwendet wird, als in der Datenbank. Insbesondere sind alle Zeitangaben in der Konfiguration auf dem Server relativ zum Testbeginn, die Services benötigen aber absolute Zeitangaben. Dies ist wichtig, um etwa einen Test oder eine Messung auf allen Observern gleichzeitig starten zu können.

3.1.2 Setup der Observer

Vor dem eigentlichen Setup wird überprüft, ob die Observer überhaupt erreichbar sind. Sollte dies nicht der Fall sein, bricht der Dispatcher ab, da der Test so nicht wie gewünscht ausgeführt werden könnte. Dann folgt das Setup der Observer, dieses ist in zwei Phasen aufgeteilt. In der ersten Phase, *Hochladen und Flashen*, werden alle Aktionen ausgeführt, welche lange dauern[1] (Grössenordnung: mehr als 1 Minute). Erst in einer zweiten Phase, *Start der Services*, werden die Services konfiguriert und gestartet. Sinn dieser Aufteilung ist, dass man nach der ersten Phase die effektive Startzeit des Tests möglichst genau abschätzen kann. In der zweiten Phase wird dann der Start des Tests und der Messungen entsprechend geplant und anschliessend die effektive Startzeit in die Datenbank eingetragen.

Das Hochladen der Dateien geschieht mittels SCP. Für das Flashen steht ein Skript auf den Observern zur Verfügung, welches den Sensorknoten einschaltet, neu programmiert und anschliessend wieder ausschaltet. Vor dem Einschalten beendet dieses auch alle eventuell noch laufenden Services, um sicherzustellen, dass nur die tatsächlich gewünschten Services laufen.

Der Benutzer des Testbeds wird darüber informiert, ob sein Test korrekt gestartet wurde oder ob ein Fehler auftrat. Dank diesem Echtzeit-Feedback kann er gegebenenfalls gleich einen neuen Test aufsetzen, falls ein Fehler auftrat.

3.1.3 Testphase

Nach Abschluss des Setups ist der Test startbereit. Der eigentliche Start wird von einem Service zeitsynchron auf allen Observern ausgeführt. Der Dispatcher muss sich also nicht selbst um den Start kümmern, sondern nur darum, den entsprechenden Service während der *Setup der Observer*-Phase korrekt zu konfigurieren. Auch um die Messdaten muss sich der Dispatcher nicht kümmern, diese werden von den Services automatisch in eine lokale Datenbank geschrieben. Die Daten in

[1] Insbesondere dauert das Flashen der Sensorknoten über 40 Sekunden lang. Es ist zudem Fehleranfällig, muss also mit einer gewissen Wahrscheinlichkeit wiederholt werden.

dieser Datenbank werden in regelmässigen Zeitabständen auf den FlockLab-Server übertragen. Wenn alles normal läuft, muss der Dispatcher also nur warten, bis der Test abgeschlossen wird.

Für den Fall, dass der Benutzer den Test abbrechen möchte, muss der Dispatcher aber schon vor dem geplanten Ende des Testes in die Aufräumphase springen.

3.1.4 Aufräumphase

Als erster Schritt wird in jedem Fall der Sensorknoten ausgeschaltet. Dies wäre zwar eigentlich nur nach einem Abbruch oder Fehler nötig. Bei fehlerfreiem Ablauf ist der Sensorknoten bereits ausgeschaltet, das stört aber nicht, da dieser den zweiten Befehl zum Abschalten schlicht ignoriert.

Damit der nächste Test wieder in einer Definierten Testumgebung laufen kann, müssen nun noch die Observer zurück in den Ausgangszustand versetzt werden. Zum einen werden alle Services gestoppt. Zum anderen werden alle temporär angelegten Dateien und Verzeichnisse wieder gelöscht, sowohl auf dem Server als auch auf den Observern. Dann trägt der Dispatcher die tatsächliche Schlusszeit in die Datenbank ein. Als letztes wartet der er noch, bis alle Daten von den Observern auf den Server übertragen wurden, warum wird im Abschnitt 3.1.7 erklärt.

3.1.5 Abschluss

Zum Schluss wird der Benutzer darüber Informiert, dass sein Test abgeschlossen (oder eben abgebrochen) wurde. Je nachdem, wie man das Abholen der Testresultate genau implementieren wird, muss der Dispatcher die Testdaten noch aus der Datenbank laden und z. B. in eine Datei schreiben.

3.1.6 Verhalten bei Fehlern

FlockLab ist ein sehr komplexes System, bei dem durchaus mal ein Fehler auftreten kann: W-LAN Verbindung kann gestört sein, beim Flashen der Sensorknoten können Fehler auftreten, eine Datenbank kann vorübergehend ausgelastet sein, usw. Der Dispatcher muss solche Fehler also erkennen und darauf reagieren können. Bei temporären Fehlern macht es am meisten Sinn, den Fehler zu ignorieren, kurz zu warten und die Aktion dann nochmals zu versuchen. Falls etwas aber mehrmals nicht klappt, kann man annehmen, dass es sich um einen permanenten Fehler handelt. In diesem Fall soll der Dispatcher den Administrator und den betroffenen FlockLab-Benutzer informieren sowie in die Aufräumphase springen.

Selbstverständlich kann auch während der Aufräumphase ein Fehler passieren. Es muss also zusätzlich darauf geachtet werden, dass der Dispatcher dann nicht nochmals in die Aufräumphase springt, ansonsten würde er in einer Endlosschleife stecken bleiben.

3.1.7 Liste von aktuell laufenden Tests

Die Observer wissen nicht, welchen Test sie gerade ausführen. Der FlockLab-Server muss daher selbst zu jeder Zeit eine aktuelle Liste besitzen, welcher Test gerade auf welchem Observer läuft. Ohne diese Liste wäre es ihm nicht möglich, die von den Observern ankommenden Messdaten dem laufenden Test zuzuordnen. Der Dispatcher hat die Aufgabe, diese Liste auf dem aktuellen Stand zu halten. Dazu wird während der Vorbereitungsphase in der Datenbank ein entsprechender Eintrag gemacht. In der Aufräumphase, nachdem die Daten aller Observer auf den Server übertragen wurden, wird der Eintrag wieder entfernt. Wie dies genau geschieht, ist im Abschnitt 3.2.2 erklärt.

3.2 Implementation

Der Dispatcher ist in der Programmiersprache Python[1] implementiert, weil die bestehenden Teile des Projekts grösstenteils in Python geschrieben sind und es so einheitlich bleibt.

Einige Funktionalitäten werden vom Dispatcher mehrmals verwendet. Es bietet sich daher an, diese als Funktionen oder als eigenständige Programme zu implementieren. Die folgende Liste gibt einen Überblick, um welche Funktionalitäten es sich handelt und wie sie implementiert wurden:

- Eine Aktion automatisch nochmals versuchen bei Fehlern, z. B. wenn ein Observer temporär nicht erreichbar ist: Funktion
- Fehlerbehandlung (Meldung an den Administrator) nach mehrmaligem Scheitern: Funktion
- Rücksetzen der Observer in den Ausgangszustand nach einem Fehler oder nachdem der Test ausgeführt wurde: Funktion
- Eine Aktion auf allen Observern (parallel) ausführen: Threads mit Warteschlange
- Kommando an einen Observer senden, Return-Wert zurück senden: Eigenes Programm (siehe 3.2.1) sowie eine Funktion im Dispatcher, damit man das Sendeprogramm bei Bedarf mühelos austauschen kann.
- Datei an Observer senden: Externes Programm (nicht selbst implementiert), ebenfalls eine Funktion im Dispatcher, damit dieses mühelos austauschen kann.
- Dem FlockLab-Benutzer oder dem Testbed Administrator eine Nachricht senden: Funktion

[1] Python Software Foundation. Python 2.5.1 bzw. 2.6.5. `http://www.python.org/`

3.2.1 Kommunikation zwischen Dispatcher und Observern

Der Dispatcher muss mehrere Male Befehle zu den Observern senden. Die wohl offensichtlichste Lösung dafür wäre, die Befehle mittels SSH zu senden, schliesslich muss für Wartungszwecke so oder so ein SSH-Daemon auf den Observern laufen. Allerdings dauert der Verbindungsaufbau über SSH relativ lange, weshalb auch die Möglichkeit evaluiert wurde, mittels des SocketServer-Moduls von Python einen eigenen Command-Daemon zu erstellen. In Tabelle 3.1 werden die Ergebnisse dieser Evaluation zusammengefasst.

SSH	eigene Software
existiert bereits (+)	muss selbst programmiert werden, mehr Entwicklungsaufwand (–)
sicher (+)	unsicher (insbesondere unverschlüsselt) (–)
langsam aufgrund des SSH Handshakes (–)	schnell, da nur ein TCP Handshake stattfindet (+)
Kommunikationsprotokoll vorgegeben (–)	kann einfach angepasst werden (+)

Tabelle 3.1: Vergleich zwischen SSH und einer selbst geschriebenen Software. (+) bedeutet Vorteil, (–) ein Nachteil.

Bei den bereits existierenden Services des FlockLab wurde nur ein minimaler Schutz implementiert, so dass z. B. nur in der Datenbank registrierte Observer Daten an den FlockLab-Server senden können. Die Kommunikation erfolgt aber jeweils unverschlüsselt. Es wird also angenommen, dass der Kommunikationskanal zwischen Server und Observern gesichert ist (z. B. mittels verschlüsseltem W-LAN oder Kabel). Daher spielt der Sicherheitsaspekt nur eine geringe Rolle für die Entscheidung. Die letzten beiden Punkte sprachen dafür, eine eigene Software zu schreiben. Bei der Programmierung des Dispatchers wurde aber darauf geachtet, dass man mit minimalem Aufwand auf SSH umstellen könnte, falls das in Zukunft nötig sein sollte.

Eine Möglichkeit, um einen Befehl zu senden, wäre gewesen, einfach das gewünschte Kommando (als String) an den Observer zu senden. Da der Dispatcher aber nur eine begrenzte Anzahl Befehle auf dem Observer ausführen muss, wurde entschieden, diese Befehle in der Datenbank des Observers abzuspeichern und nur eine Befehlsnummer (command ID) an den Observer zu senden. Dies macht zwar die Implementation etwas mühsamer, hat aber mehrere Vorteile: Auf verschiedenen Observern können verschiedene Software-Versionen laufen, welche nicht zwingend die gleiche Syntax haben müssen. Mit grosser Sicherheit wird zudem ein Befehl nie zu lang für ein TCP- oder UDP-Paket. Diese Implementation bietet auch noch ein minimales Mass an Sicherheit, da ein möglicher Angreifer nicht beliebige Befehle auf den Observern ausführen kann, sondern nur die vom FlockLab erlaubten Befehle. Eine Übersicht aller benötigten Befehle befindet sich in der Tabelle 3.2.

Nr.	Befehl	Argument(e)
1	flocklab_isdb_empty.py	
2	flocklab_reprog_tg.py	--image=<filepath>
3	flocklab_stp_servs.py	
11	tg_on.sh	
12	tg_off.sh	
13	tg_reset.sh	
21	ensure_dir.py	--path=<path>
22	rm	<path>
23	rmdir	<path>
100	flocklab_gpiomonitor	--quiet -start --database
101	flocklab_gpiomonitor	--quiet -stop
102	flocklab_gpiomonitor	--quiet -addbatch <arguments>
103	flocklab_gpiomonitor	--quiet -add <filepath>
110	flocklab_gpiosetting	--quiet -start --database
111	flocklab_gpiosetting	--quiet -stop
112	flocklab_gpiosetting	--quiet -addbatch <arguments>
113	flocklab_gpiosetting	--quiet -add <filepath>
120	flocklab_powerprofiling	--quiet -start --database
121	flocklab_powerprofiling	--quiet -stop
122	flocklab_powerprofiling	--quiet -addbatch <arguments>
123	flocklab_powerprofiling	--quiet -add <filepath>
131	flocklab_serial_reader.py	--quiet --stop
133	flocklab_serial_reader.py	--targetos=<os> --dbpath=<filepath> <more arguments>

Tabelle 3.2: Liste von Befehlen, welche der Dispatcher auf den Observern ausführen kann

Die Kommunikation geschieht mittels eines Empfänger-Daemons auf den Observern und eines Sende-Clients auf dem FlockLab-Server. Der Sende-Client wird vom Dispatcher aufgerufen, sendet das Kommando und empfängt den Rückgabewert des ausgeführten Programmes und des Empfänger-Daemons. Je nachdem, ob die Ausführung erfolgreich war oder scheiterte, gibt er einen Fehlerwert zurück an den Dispatcher. Sender und Empfänger kommunizieren über TCP. Trotz des höheren Overheads ist TCP in diesem Fall sinnvoller als UDP, da die Anfrage und Antwort zur selben TCP Verbindung gehören und somit eindeutig einander zugeordnet werden können.

Das Kommunikationsprotokoll wurde folgendermassen festgelegt:

```
Anfrage, vom Sender an den Empfänger:
(command_id, arguments)

Antwort, vom Empfänger zurück an den Sender:
(cmdreceiverd_statuscode, cmd_statuscode)
```

Dabei ist *command_id* eine Ganzzahl und entspricht der Befehlsnummer, *arguments* sind die Argumente des Befehls (insbesondere der Pfad zum Batch-File) als (Python-) Tupel. Die beiden Werte in der Antwort sind ebenfalls Ganzzahlen, *cmd_statuscode* ist der Rückgabewert des aufgerufenen Befehls und *cmdreceiverd_statuscode* der Rückgabewert des Empfänger-Daemons selbst.

3.2.2 Liste der laufenden Tests

Wie im Abschnitt 3.1.7 erwähnt, muss der Server wissen, welcher Observer gerade welchen Test ausführt. Dazu wurde in der Datenbank für jeden Test ein neues Feld namens *is_running* angelegt. Dieses Feld muss genau zwei Zustände speichern können, nämlich *Test läuft momentan* oder *Test läuft nicht*. In der Datenbank ist auch eine Zuordnungstabelle Test / benötigte Observer abgespeichert. Da ein Observer höchstens einen Test gleichzeitig ausführt, lässt sich mit Hilfe dieser Zuordnungstabelle und des *is_running*-Feldes eindeutig feststellen, zu welchem Test die ankommenden Messdaten eines bestimmten Observers gehören.

Kapitel 4

Scheduler

Der Scheduler hat die Aufgabe, zu planen, wann welcher Test ausgeführt werden soll. Zur geplanten Zeit soll er den Test vom Dispatcher ausführen lassen.

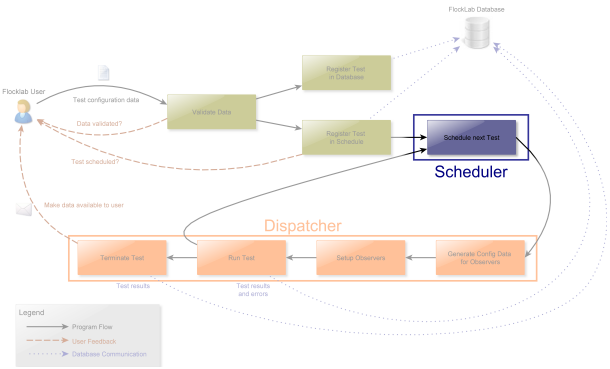


Abbildung 4.1: Der Scheduler im gesamten Testablauf.

4.1 Allgemeine Überlegungen

Um mehrere Tests zu bestimmten Zeiten automatisch auf dem Testbed ausführen zu können, muss ein Ablaufplan erstellt werden, welcher Test wann dran kommt. Dieser Ablaufplan soll fair sein, es soll also kein Test vernachlässigt werden. Dies ist die Aufgabe des Schedulers. Er plant, welcher Job wann ausgeführt werden soll. Rechtzeitig vor der geplanten Startzeit ruft er den Dispatcher auf, damit dieser den Test aufsetzen und pünktlich starten lassen kann.

Für das weitere Design des Schedulers wurden folgende Annahmen getroffen:

- Ein "Job" ist ein Test, welcher ein- oder mehrmals auf dem Testbed ausgeführt wird.

- Tests können nicht unterbrochen werden. Wenn ein Test abgebrochen wurde, muss er wieder von vorne starten.
- Jobs (und somit auch Tests) sind nicht unendlich lang.

4.2 Mögliche Benutzerwünsche und weitere Ideen

Eine Auflistung von möglichen Benutzerwünschen sowie Ideen für Lösungsansätze sind in der Tabelle 4.1 zu finden. Es folgt eine Liste von weiteren Überlegungen, die den Scheduler betreffen:

- "Orthogonal" zu den in der Tabelle angegebenen Szenarien (ausser (2)) könnte man jedem Job noch eine Priorität zuordnen, als Beispiel n Prioritätsstufen, 1. Priorität (höchste) bis n-te Priorität (niedrigste). Diese Jobs kommen dann in n unabhängige "Pools". Erst wenn im n-ten Pool keine ausführbaren Jobs mehr sind, werden solche aus dem (n+1)-ten Pool ausgewählt.
- Die Länge eines Jobs kann theoretisch unbeschränkt sein, so lange es keine anderen Jobs gibt, stört das auch nicht. Aus praktischer Hinsicht macht es aber durchaus Sinn, eine Obergrenze festzulegen (vor Allem für Jobs mit hoher Priorität).
- Wenn ein Job mit niedriger Priorität ausgeführt wird und in der Zwischenzeit ein neuer Job mit höherer Priorität hinzukommt, würde üblicherweise der Job mit niedriger Priorität abgebrochen und zuerst der neue ausgeführt. Eventuell macht es Sinn, hier über Ausnahmen nachzudenken (z. B. wenn der niedrigpriorisierte Test schon mehrere Stunden lief und in wenigen Minuten beendet wäre).
- Der Scheduler muss genügend Reserve-Zeit zwischen einzelnen Tests einplanen, während der der Dispatcher seine Arbeit verrichten kann.

4.3 Mögliche Implementation

Eine mögliche Implementation des beschrieben Szenarios (in Pseudocode) ist im Anhang B dargestellt. Obwohl die Architektur des FlockLabs grundsätzlich mehrere FlockLab-Server unterstützt, wäre es sinnvoll, den Scheduler auf dem selben FlockLab-Server auszuführen, wie den Dispatcher. Dies erleichtert das Aufrufen des Dispatchers durch den Scheduler.

Es wurde im Verlauf der Arbeit recht bald klar, dass die Zeit nicht reichen wird, um sowohl den Scheduler als auch den Dispatcher zu implementieren. Deshalb konzentrierte ich mich auf das Implementieren des Dispatchers, beim Scheduler blieb es bei dieser theoretischen Betrachtung.

Mögliche Wünsche des Benutzers	Idee, Lösungsansatz
Genauer Zeitpunkt (1)	Der Benutzer kann das Testbed für eine bestimmte Zeit reservieren. Falls diese Zeit schon vergeben ist (FCFS), wird ihm, direkt bei der Reservation, eine entsprechende Fehlermeldung angezeigt.
	Diese Jobs haben höchste Priorität, weil die Zeit "versprochen" wurde und nun nicht verschoben werden kann.
	Um Missbrauch zu vermeiden, kann ein Benutzer nicht unbeschränkt viele solcher Jobs in Auftrag geben.
x-mal, jeweils zur gleichen Tageszeit (2)	Zum reservierten Zeitpunkt wird versucht, den Test auszuführen. Falls das Testbed (von Jobs mit höherer Priorität) besetzt ist, wird der Job um einen Tag verschoben.
	Falls mehrere Jobs im gleichen Zeitraum ausgeführt werden sollen, sollte Round Robin verwendet werden, d.h. es sollte an jedem Tag ein anderer Job dran kommen (FCFS wäre in dem Fall wohl doch etwas zu unfair).
	Neue Jobs werden grundsätzlich "hinten angehängt" (FCFS). Als Erweiterung könnte man noch vorsehen, dass Jobs, welche noch nie ausgeführt wurden, über solche priorisiert werden, die bereits mindestens 1 Mal ausgeführt wurden.
Genauer Zeitpunkt, Datum egal (3)	Spezialfall von (2) mit x=1
x-mal, in einem bestimmten Zeitraum (nachts, am Wochenende, ...) (4)	Im reservierten Zeitraum werden die Jobs ausgeführt, ausser wenn das Testbed (von Jobs mit höherer Priorität) besetzt ist. Wenn der Zeitraum vorbei ist, ohne dass der Job ausgeführt werden konnte, wird er auf den nächsten möglichen Zeitraum verschoben.
	Der Rest ist analog zu (2) bzw. (3).
	Diese Jobs sollten niedrigere Priorität haben, als diejenigen aus (1), da sie einfacher verschoben werden können.
x-mal, egal wann (5)	Wie (2), nur eben ohne Einschränkung auf eine bestimmte Zeit.
	Diese Jobs können am einfachsten verschoben werden und sollten demzufolge die niedrigste Priorität haben.
irgendwann, ein Mal (6)	Spezialfall von (5) mit x=1

Tabelle 4.1: Scheduler: Mögliche Benutzerwünsche und Lösungsansätze

Kapitel 5

Fazit und zukünftige Arbeiten

5.1 Fazit

Die Semesterarbeit beihnaltete sowohl den theoretischen Entwurf als auch die Implementation der Software. Es wurde im Verlaufe der Arbeit recht bald klar, dass es nicht für alles reichen würde. Deshalb wurde entschieden, dass ich mich auf den Entwurf und die Implementation des Dispatchers konzentrieren soll.

5.1.1 Scheduler

Der Scheduler wurde in Kapitel 4 theoretisch behandelt. Ob die dort angegebene "Wunschliste" tatsächlich den realen Wünschen der zukünftigen FlockLab-Benutzer entspricht, ist jedoch sehr von deren Forschungen abhängig. Ich schlage daher vor, dies vor der Implementation z. B. mit einer Umfrage heraus zu finden.

Der im Anhang B beschriebene Algorithmus ist als Entwurf anzusehen und kann noch Fehler enthalten.

Wenn mehrere Tests nicht alle Knoten des Testbeds benutzen, könnte man diese gleichzeitig ausführen. Falls diese Situation im FlockLab häufig vorkommt, müsste man sich über diesen Aspekt wohl noch weitere Gedanken machen. Der Scheduling Algorithmus müsste entsprechend angepasst werden, damit er solche Jobs findet und gleichzeitig plant.

5.1.2 Dispatcher

Das Grundgerüst des Dispatchers wurde implementiert und ausgiebig getestet. Aus Zeitgründen konnten jedoch nicht mehr alle Funktionalitäten implementiert werden. Die betroffenen Stellen sind im Code jeweils mit einem "TODO" markiert und beschrieben, was noch zu tun ist. Dies betrifft insbesondere die Datenbankabfragen und den Parser zu Beginn des Dispatchers. An Stelle derer Antworten wurde ein Test-Fall fest programmiert, um die übrigen Funktionalitäten des Dispatchers (Verteilen und starten der Jobs usw.) überhaupt testen zu können.

FlockLab unterstützt mehrere Plattformen, d. h. Typen von Sensorknoten. Der Dispatcher unterstützt momentan aber nur die TinyNode-Plattform. Dies liegt daran, dass Befehle wie *tg_on*, *reprog_tg*, usw. nur dafür existieren. Mein Vorschlag ist, bei diesen Plattform-abhängigen Befehlen ein Argument --platform=... einzuführen, mit dem der Befehl für die gewünschte Plattform ausgeführt wird. Anschliessend müsste der Dispatcher entsprechend ergänzt werden, so dass er das platform Argument bei diesen Befehlen mitsendet.

5.2 Integration in das bestehende System

Damit die neu geschriebene Software läuft, müssen die erstellten Skripts an die richtigen Orte kopiert, sowie die Datenbanken angepasst werden. Es folgt eine kleine Installationsanleitung.

5.2.1 Observer

Der Empfänger-Daemon *rpc_cmdreceiverd.py* und das Tool *ensure_dir.py* müssen ins Verzeichnis */usr/bin* kopiert werden. Bei den Benutzerrechten muss ggf. das "ausführbar"-Bit gesetzt werden. Der Empfänger-Daemon sollte so eingerichtet werden, dass er automatisch mit dem Betriebssystem des Observers gestartet wird.

Die Liste der Befehle für den Empfänger-Daemon ist in der Datei *command_list_for_cmdreceiverd.sql* gespeichert. Damit dieser seine Befehle aber auch tatsächlich findet, muss die Liste noch in die bestehende Datenbank eingefügt werden. Dies kann mit dem Befehl

```
sqlite -init command_list_for_cmdreceiverd.sql
    /home/root/mmc/flocklab/flocklab_main.sdb
```

erfolgen.

Damit Dateien mittels SCP hochgeladen werden können, muss der öffentliche Schlüssel des Servers auf allen Observern in der *.ssh/authorized_keys*-Liste eingetragen werden. Zudem darf der private Schlüssel auf dem Server nicht mit einem Passwort geschützt sein (bzw. es muss ein ssh-agent entsprechend eingerichtet werden), so dass das Hochladen auch ohne Eingabe eines Passworts funktioniert.

5.2.2 FlockLab-Server

Auf dem FlockLab-Server muss der Dispatcher *flocklab_serv_dispatcher.py* sowie der Sender-Client *rpc_cmdsender.py* ins Verzeichnis */usr/bin* kopiert werden. In der FlockLab-Datenbank muss die Tabelle *tbl_serv_observer* um das Feld *port_cmdreceiverd* ergänzt werden. Dies geschieht mit dem SQL-Befehl

```
ALTER TABLE 'tbl_serv_observer' ADD 'port_cmdreceiverd'
    INT NOT NULL DEFAULT '<port>' AFTER 'port_datarecvd'
```

wobei <port> durch die gewünschte Nummer des Standard-Ports ersetzt wird. In der Tabelle *tbl_serv_tests* ist ein neues Feld *is_running* einzufügen, welches darstellt, ob der Test gerade läuft oder nicht. Dieses Feld kann mittels

```
ALTER TABLE 'tbl_serv_tests' ADD 'is_running' TINYINT NOT
    NULL DEFAULT '0' AFTER 'max_duration'
```

erstellt werden.

5.3 Dokumentation, weitere Anleitungen

Eine Anleitung zur Benutzung der erstellten Software ist in dieser Arbeit bewusst keine erwähnt. Man erhält nähere Informationen zur Benutzung aller Programme, wenn man den jeweiligen Befehl mit dem Argument --help eingibt. Die implementierten Funktionen des Dispatchers entsprechen, soweit dies möglich war, den Loops und funktionalen Blöcken des vorgestellten Flussdiagramms. Bei der Programmierung wurde darauf geachtet, dass der Code gut kommentiert ist, daher wird hier nicht näher auf den Programmcode eingegangen. Namen für neu erstellte Programme oder Felder in der Datenbank wurden jeweils nach dem bereits vorhandenen Benennungs-Schema vergeben.

Die im Rahmen dieser Arbeit geschriebene Software befindet sich auf der beiliegenden CD-ROM. Falls die CD-ROM verloren gegangen ist, kann die Software auch beim Autor per E-Mail angefortert werden: <kronigs@ee.ethz.ch> oder <stefan.kronig@gmail.com>.

Literaturverzeichnis

[1] Christoph Walser. Wireless sensor network testbed 2.0: A new service oriented architecture. Master thesis, ETH Zurich, Switzerland, Computer Engineering Lab, 2009.

[2] Jan Beutel, Roman Lim, Andreas Meier, Lothar Thiele, Christoph Walser, Matthias Woehrle, and Mustafa Yuecel. The flocklab testbed architecture. In *SenSys '09: Proceedings of the 7th ACM Conference on Embedded Networked Sensor Systems*, pages 415–416, New York, NY, USA, 2009. ACM.

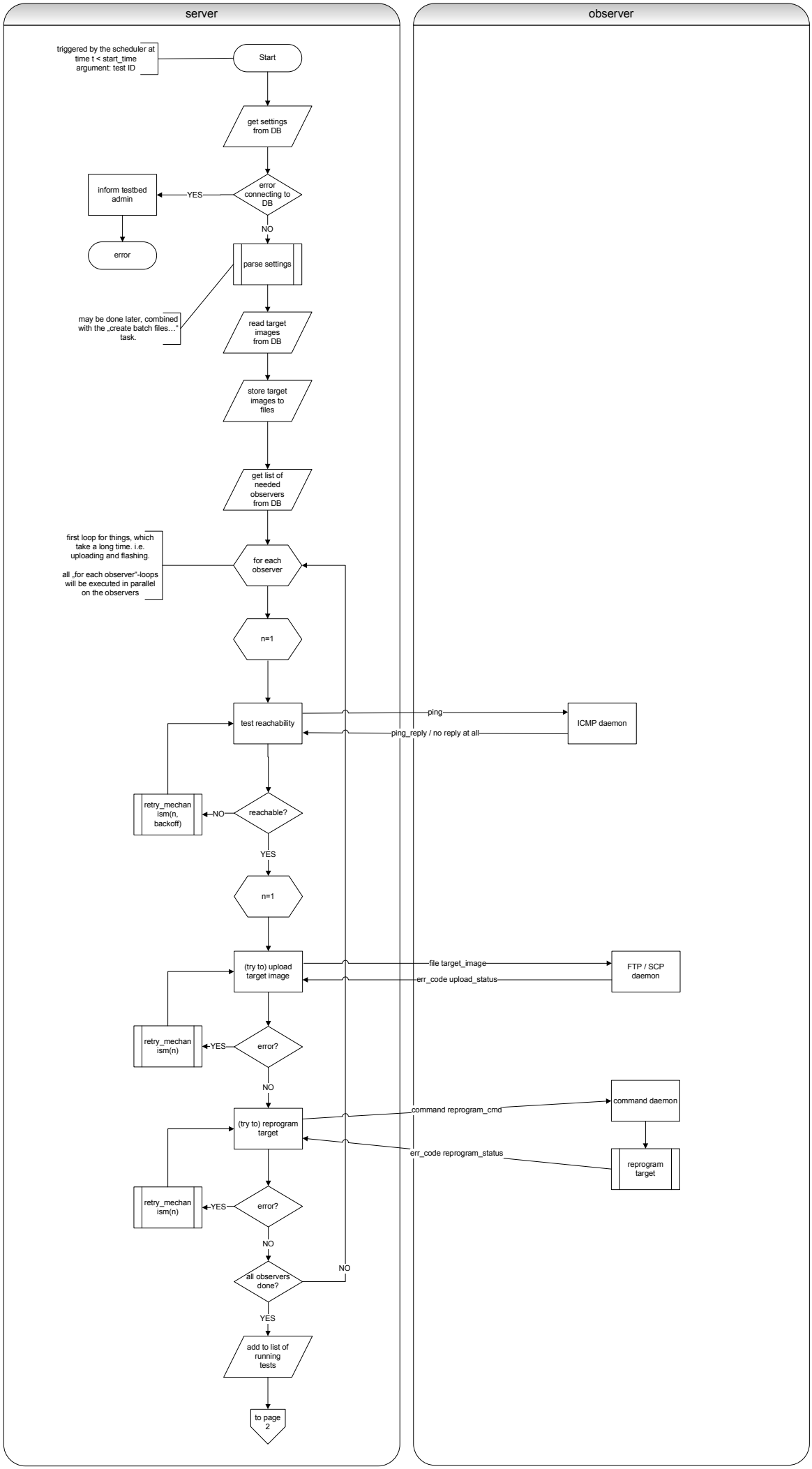
[3] Matthias Dyer, Jan Beutel, Thomas Kalt, Patrice Oehen, Lothar Thiele, Kevin Martin, and Philipp Blum. Deployment support network - a toolkit for the development of WSNs. In *EWSN '07: Proceedings of the 4th European Workshop on Sensor Networks*, pages 195–211. Springer, 2007.

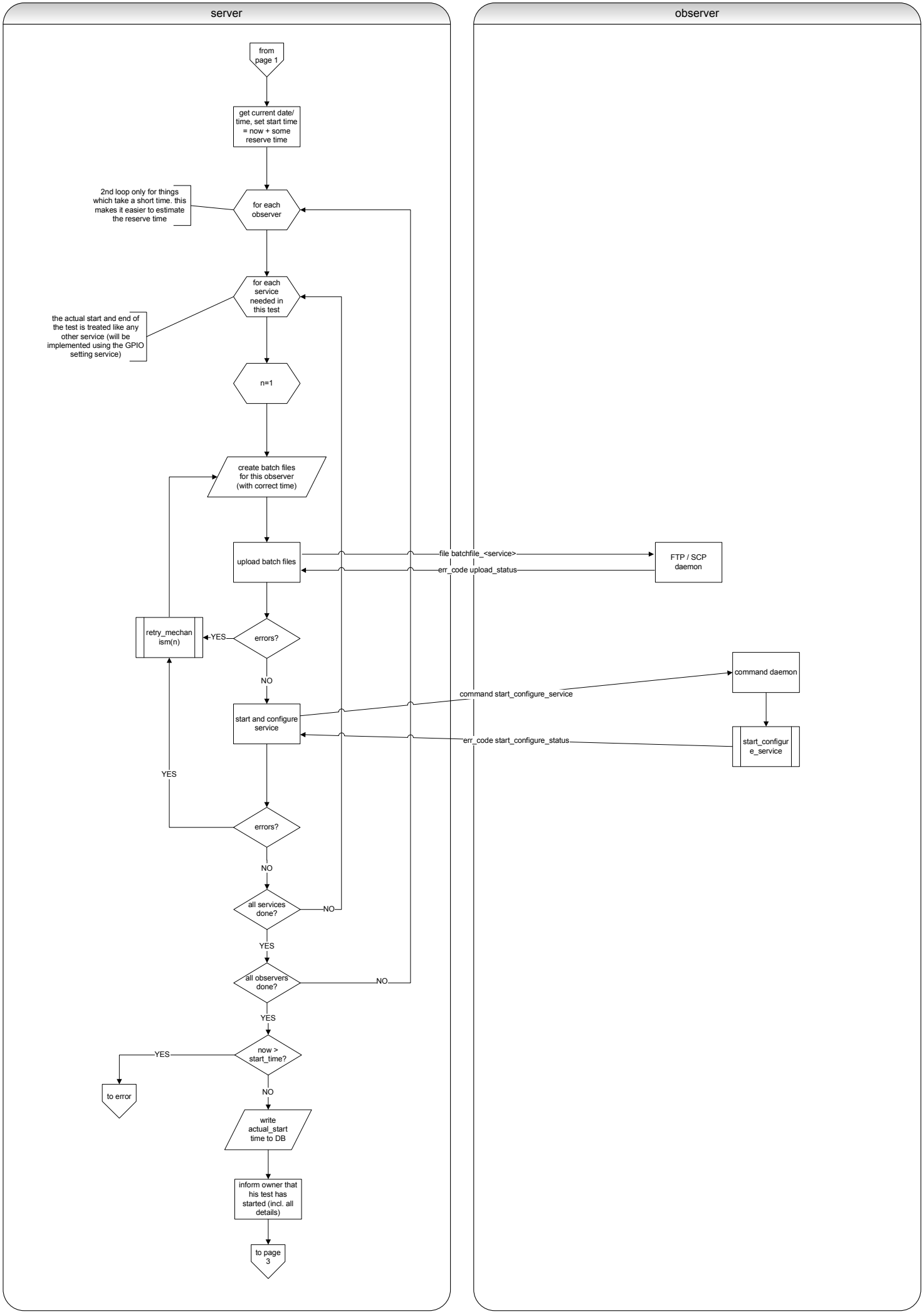
[4] Geoffrey Werner-Allen, Patrick Swieskowski, and Matt Welsh. MoteLab: a wireless sensor network testbed. In *IPSN '05: Proceedings of the 4th international symposium on information processing in sensor networks*, page 68, Piscataway, NJ, USA, 2005. IEEE Press.

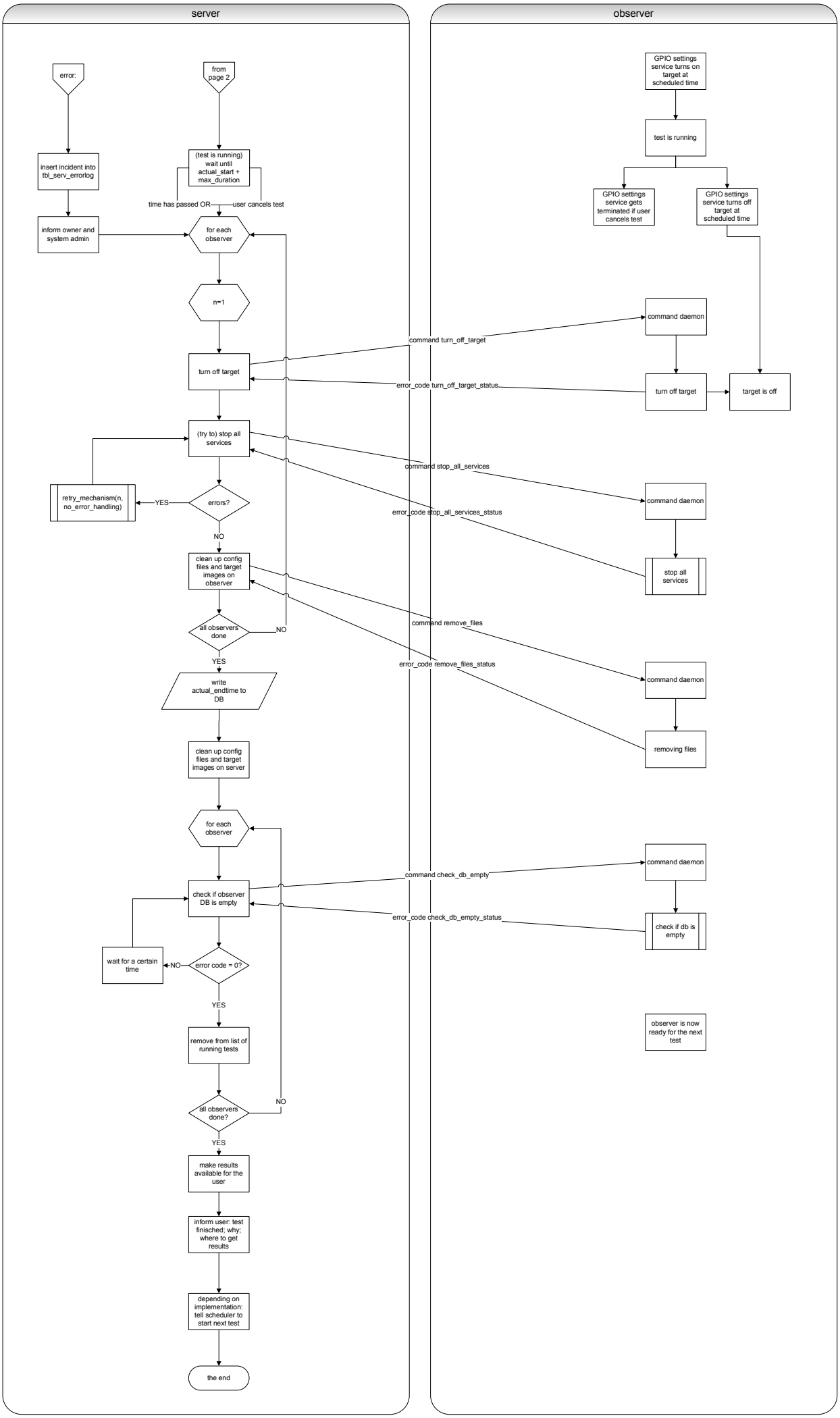
[5] B. N. Chun, P. Buonadonna, A. AuYoung, Chaki Ng, D. C. Parkes, J. Shneidman, A. C. Snoeren, and A. Vahdat. Mirage: a microeconomic resource allocation system for sensornet testbeds. In *EmNets '05: Proceedings of the 2nd IEEE workshop on Embedded Networked Sensors*, pages 19–28, Washington, DC, USA, 2005. IEEE Computer Society.

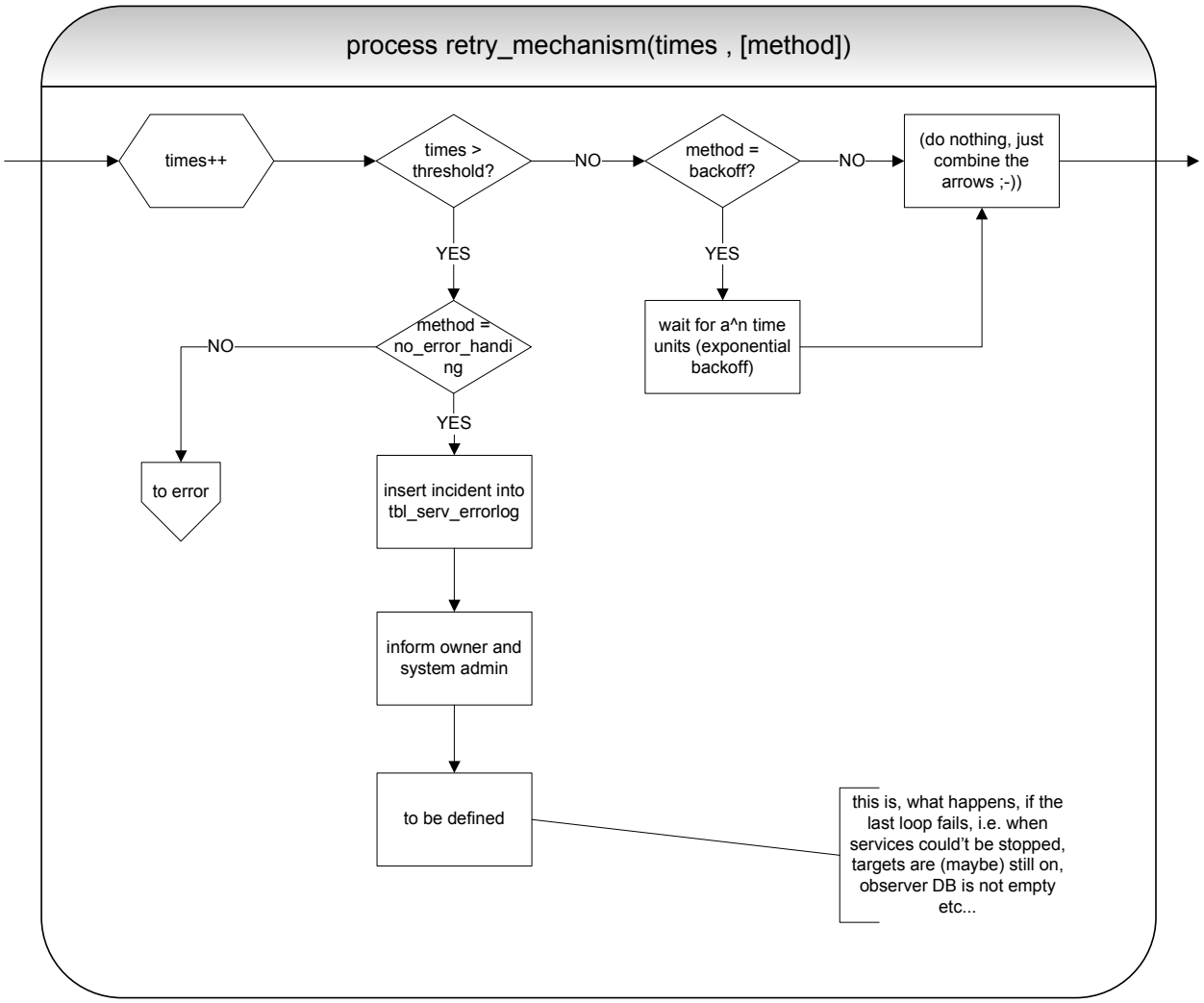
Anhang A

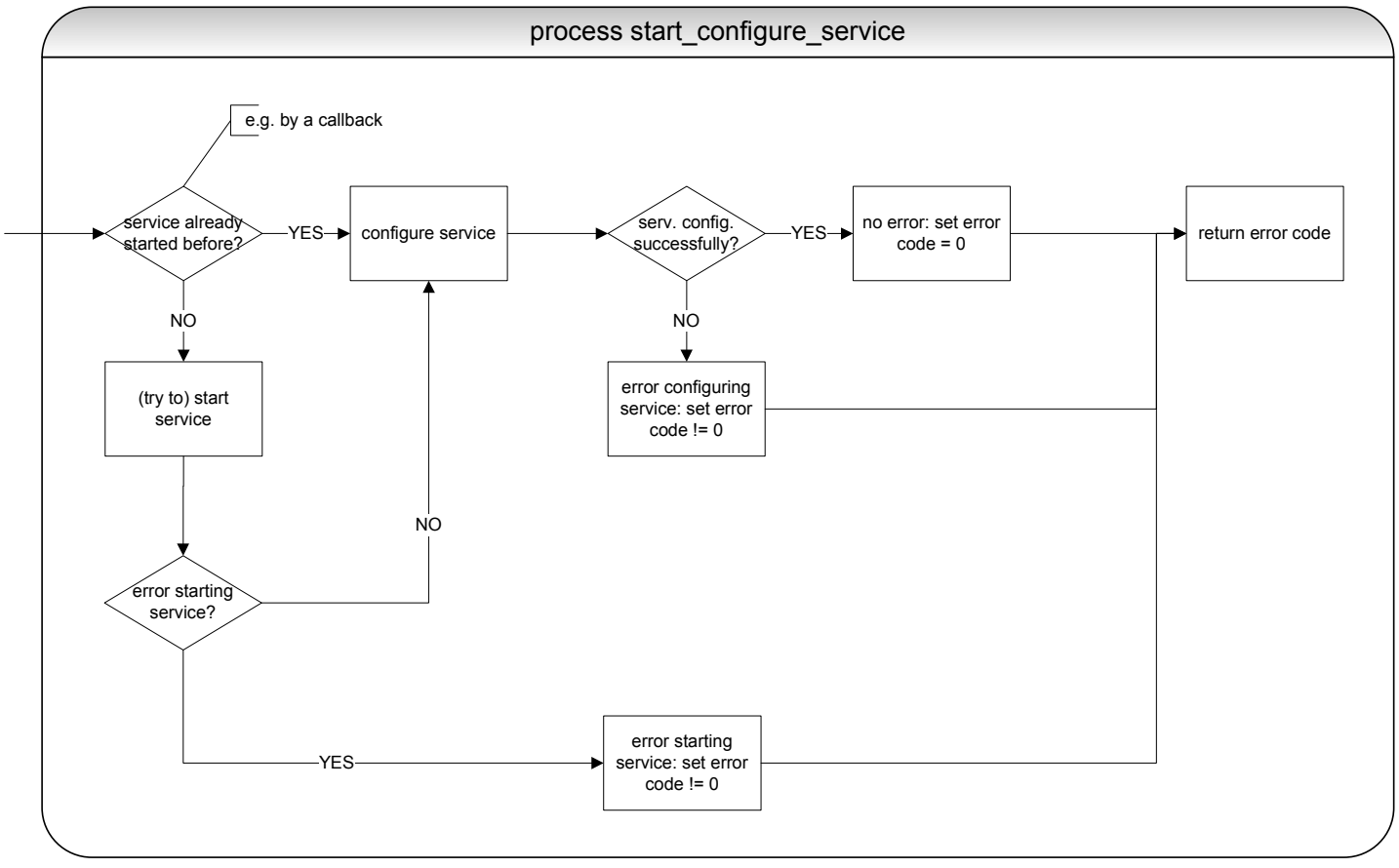
Detaillierter Testablauf


server
observer
triggered by the scheduler at time t < start_time argument: test ID
Start
get settings from DB
error connecting to DB
YES
inform testbed admin
error
NO
parse settings
may be done later, combined with the „create batch files…“ task.
read target images from DB
store target images to files
get list of needed observers from DB
first loop for things, which take a long time. i.e. uploading and flashing.
all „for each observer“-loops will be executed in parallel on the observers
for each observer
n=1
test reachability
ping
ping_reply / no reply at all
ICMP daemon
retry_mechanism(n, backoff)
NO
reachable?
YES
n=1
(try to) upload target image
file target_image
err_code upload_status
FTP / SCP daemon
retry_mechanism(n)
YES
error?
NO
command reprogram_cmd
command daemon
(try to) reprogram target
err_code reprogram_status
reprogram target
retry_mechanism(n)
YES
error?
NO
all observers done?
NO
YES
add to list of running tests
to page 2


server
observer
from page 1
get current date/ time, set start time = now + some reserve time
2nd loop only for things which take a short time. this makes it easier to estimate the reserve time
for each observer
for each service needed in this test
the actual start and end of the test is treated like any other service (will be implemented using the GPIO setting service)
n=1
create batch files for this observer (with correct time)
upload batch files
file batchfile_<service>
err_code upload_status
FTP / SCP daemon
errors?
YES
retry_mechanism(n)
NO
command daemon
command start_configure_service
start and configure service
err_code start_configure_status
start_configure_service
YES
errors?
NO
all services done?
NO
YES
all observers done?
NO
YES
now > start_time?
YES
to error
NO
write actual_start time to DB
inform owner that his test has started (incl. all details)
to page 3


server
observer
error:
from page 2
insert incident into tbl_serv_errorlog
(test is running) wait until actual_start + max_duration
time has passed OR
user cancels test
inform owner and system admin
for each observer
n=1
turn off target
(try to) stop all services
retry_mechanism(n, no_error_handling)
YES
errors?
NO
clean up config files and target images on observer
all observers done
NO
YES
write actual_endtime to DB
clean up config files and target images on server
for each observer
check if observer DB is empty
wait for a certain time
NO
error code = 0?
YES
remove from list of running tests
all observers done?
NO
YES
make results available for the user
inform user: test finisched; why; where to get results
depending on implementation: tell scheduler to start next test
the end
GPIO settings service turns on target at scheduled time
test is running
GPIO settings service gets terminated if user cancels test
GPIO settings service turns off target at scheduled time
command daemon
command turn_off_target
turn off target
target is off
error_code turn_off_target_status
command stop_all_services
command daemon
error_code stop_all_services_status
stop all services
command remove_files
command daemon
error_code remove_files_status
removing files
command daemon
command check_db_empty
check if db is empty
error_code check_db_empty_status
observer is now ready for the next test


process retry_mechanism(times , [method])
times++
times > threshold?
NO
method = backoff?
NO
(do nothing, just combine the arrows ;-))
YES
YES
method = no_error_handi ng
NO
wait for a^n time units (exponential backoff)
YES
to error
insert incident into tbl_serv_errorlog
inform owner and system admin
to be defined
this is, what happens, if the last loop fails, i.e. when services could't be stopped, targets are (maybe) still on, observer DB is not empty etc...


process start_configure_service
e.g. by a callback
service already started before?
YES
configure service
serv. config. successfully?
YES
no error: set error code = 0
return error code
NO
NO
(try to) start service
error configuring service: set error code != 0
NO
error starting service?
YES
error starting service: set error code != 0

Anhang B

Möglicher Ablauf des Schedulers

in Pseudocode:

```
# first , finish running job
if running_job has reached its maximum runtime:
        abort this job
        increment the counter , how many times it has run
        if counter >= max. number of runs:
                delete this job from queue


# exact date and time , these jobs have highest priority
if exact_date_and_time_job is running:
        # do nothing
        exit


if exact_date_and_time_job wants to be executed now:
        abort all running jobs # as the testbed was
           explicitely reserved for this job
        start this job
        exit


for each priority p, from 1 to n:
        # exact time , within p, these jobs have highest
           priority
        if exact_time_job of priority p is running:
                # do nothing
                exit
```

```
        if exact_time_job of priority p (let's call it j)
           wants to be executed:
                find all other exact_time_jobs of priority
                   p which want to be executed before j
                   ends
                        # i.e. started before and ended
                           after j ends
                let round-robin decide, which of these
                   jobs should be executed next
                if round robin selected j:
                        abort all running jobs
                        start j
                else: # i.e. round robin selected another
                   job
                        # do nothing, right now, RR will
                           select this job later
        exit;


        analogue for time_interval_jobs;

        analogue for no_matter_when_jobs;

# end for each priority
```

Anhang C

Aufgabenstellung

ETH
Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

Computer Engineering and Networks Lab (TIK)

Spring Term 2010

SEMESTER THESIS

for
Stefan Kronig

Supervisor: Matthias Woehrle
Co-Supervisor: Christoph Walser

Start date: 22 March 2010
End date: 25 June 2010

Management Software for Wireless Sensor Network Testbed

Introduction

Wireless Sensor Network (WSN) nodes are small battery-powered platforms usually equipped with a micro controller, a radio module and a sensor. These nodes can, for instance, be deployed on a mountain to measure data (i.e. temperature) and forward this data to a base station[1].

The battery-powered nodes are not only constrained by the very limited power supply (battery) but also in processing power, memory size and communication possibilities to name a few. These limitations result in various difficulties when implementing a WSN application. Especially the combination of low-power operation and wireless communication seems to be a crux for the robustness and reliability of WSNs. In order to arrive at a functioning system at deployment time, several test platforms have been proposed such as simulators [2, 3], emulators [4] or various testbeds [5, 6, 7].

With FlockLab [8] we are currently developing a new, service-oriented testbed architecture which allows for detailed monitoring and stimulation of wireless sensor nodes. In particular, time-accurate state extraction and power measurements are provided in a distributed, yet synchronized context.

FlockLab brings the functionalities of expensive lab instruments into a distributed context for detailed testing of a sensor network. This is achieved by pairing a sensor node with dedicated hardware for monitoring and stimulation. Different services such as measuring power consumption and time accurate pin monitoring and setting are provided to the tester. By reducing accuracy of measurements to a sufficient level (e.g., tens of μs temporal granularity across nodes), costs are significantly reduced to enable an affordable distributed lab instrument. Hence, in difference to previous testbeds, which only allowed for detailed measurements on individual nodes, numerous nodes can be observed in detail. Test configurations as well as measurement data are stored in several databases distributed over the test nodes and a server.

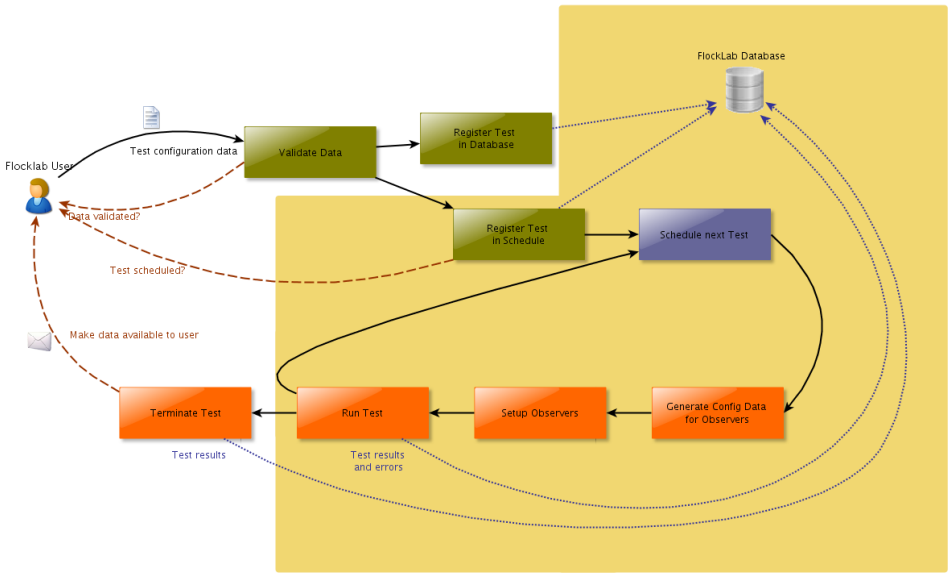


Figure 1: Overview of test management SW. A user provides configuration data, which needs to be evaluated. After validation, a test is scheduled by the test management software. On its scheduled slot, a test is dispatched onto the testbed. After the test is completed (or on a fail), the user is notified that the test has finished. The marked area depicts the focus of this thesis.

Problem Definition

The goal of this semester thesis is to design and implement a test management software for FlockLab. The software should handle all tasks needed to schedule, run and evaluate tests on the testbed according to Figure 1. A major goal is to find an efficient and fair scheduling algorithm for scheduling testbed jobs from multiple users.

This work particularly focusses on:

- A control algorithm for the individual management tasks,
- an analysis and definition of requirements of a scheduler for testbed jobs from multiple users resulting in representative testcases,
- an efficient scheduling of testbed jobs from multiple users [9, 5, 10],
- visualization of test status and results for the user.

In order to reach these goals, the project should proceed according to the following steps:

1. The student should write a project plan and identify its milestones (thematically and concerning time). In particular there should be enough room for the final presentation and the report.

2. The student should study related work in the area of testbeds and resource management [11, 6, 7], focusing on scheduling solutions such as [9, 5, 10] or Condor[1]. The student should also look out for further related work in this area. The results of this literature research should be written down as a first chapter of the report.
3. The student should evaluate and compare different scheduling algorithms.
4. The student should design and implement software for scheduling testbed jobs as well as control algorithms for the following management tasks according to Figure 1:
   - Generation of configuration data for observers
   - Setting up of observers
   - Running a test
   - Presentation of test results to the user

   Note that these tasks assume the existence of valid jobs in the database. Hence the student needs to design testcases which populate the database accordingly. Additionally, the testcases should include a specification of requirements concerning the scheduling. These requirements are a result of the initial literature study. If time permits the tasks of test registration, i. e., the adding of test data into the database may be considered.
5. The student should test the implementation and evaluate its correctness and performance.

Organization

- Duration of the Work:
  This Semester Thesis starts 22 March 2010 and has to be finished no later than 25 June 2010.
- Project Plan:
  A project plan with its milestones is held and updated continuously. Unforeseen difficulties that change the project plan have to be documented and should be discussed with the supervisors.
- Weekly Meetings/Reports:
  In regular (weekly) meetings with the supervisors, the current state of the work, potential difficulties as well as future directions are discussed. The day before the weekly meeting a brief status report should be sent to the supervisors commenting on these issues, in order to allow an adequate meeting preparation for the student and the supervisors.
- Research Journal:
  The work's progress is written down in a research journal that is handed in to the supervisor at the end of the project.
- Beginners Presentation:
  Approximately two to three weeks after the start the student will shortly present the objectives of the work as well as some background on the topic. The presentation should be no longer than 5 minutes and consist of maximally two slides.
- Final Presentation:
  By the end of the project, the student will present the achieved result. The presentation should not exceed 15 minutes.
- Documentation:
  At the end of the project, no later than 25 June 2010, the student will have to hand in a written report. Together with the system implementation/software this report is the main outcome of the project. Document your work accurately. Additionally make sure to comment your code extensively, allowing a follow-up project.

[1] http://www.cs.wisc.edu/condor/

- Evaluation of the work:
  The criteria for grading the work are described in [12].
- Finishing up:
  The required resources should be cleaned up and handed back in.

References

[1] J. Beutel, S. Gruber, A. Hasler, R. Lim, A. Meier, C. Plessl, I. Talzi, L. Thiele, C. Tschudin, M. Woehrle, and M. Yuecel, "PermaDAQ: A scientific instrument for precision sensing and data recovery in environmental extremes," in ACM/IEEE International Conference on Information Processing in Sensor Networks (IPSN 2009), p. (to appear), 2009.

[2] P. Levis, N. Lee, M. Welsh, and D. Culler, "TOSSIM: Accurate and scalable simulation of entire TinyOS applications," in Proc. 1st ACM Conf. Embedded Networked Sensor Systems (SenSys 2003), pp. 126–137, Nov. 2003.

[3] L. Girod, N. Ramanathan, J. Elson, T. Stathopoulos, M. Lukac, and D. Estrin, "Emstar: A software environment for developing and deploying heterogeneous sensor-actuator networks," ACM Trans. Sen. Netw., vol. 3, no. 3, p. 13, 2007.

[4] B. L. Titzer, D. K. Lee, and J. Palsberg, "Avrora: scalable sensor network simulation with precise timing," in Proc. 4th Int'l Conf. Information Processing Sensor Networks (IPSN '05), p. 67, 2005.

[5] G. Werner-Allen, P. Swieskowski, and M. Welsh, "MoteLab: A wireless sensor network testbed," in Proc. 4th Int'l Conf. Information Processing Sensor Networks (IPSN '05), pp. 483–488, Apr. 2005.

[6] V. Handziski, A. Koepke, A. Willig, and A. Wolisz, "Twist: a scalable and reconfigurable testbed for wireless indoor experiments with sensor networks," in Proc. 2nd international workshop on Multi-hop ad hoc networks: from theory to reality (REALMAN '06), (New York, NY, USA), pp. 63–70, ACM Press, 2006.

[7] M. Dyer, J. Beutel, L. Thiele, T. Kalt, P. Oehen, K. Martin, and P. Blum, "Deployment support network - a toolkit for the development of WSNs," in Proc. 4th European Workshop on Sensor Networks (EWSN 2007), pp. 195–211, 2007.

[8] J. Beutel, R. Lim, A. Meier, L. Thiele, C. Walser, M. Woehrle, and M. Yuecel, "Poster abstract: The flocklab testbed architecture," in Proc. 7th ACM Conf. Embedded Networked Sensor Systems (SenSys 2009), (Berkeley, CA, USA), pp. 415–416, November 2009.

[9] B. Chun, P. Buonadonna, A. AuYoung, C. Ng, D. Parkes, J. Shneidman, A. Snoeren, and A. Vahdat, "Mirage: A microeconomic resource allocation system for sensornet testbeds," in Proc. 2nd IEEE Workshop on Embedded Networked Sensors (EmNetS-II), p. to appear, IEEE, Piscataway, NJ, May 2005.

[10] M. Günes, B. Blywis, and F. Juraschek, "Concept and design of the hybrid distributed embedded systems testbed," Tech. Rep. B 08-10, Institute of Computer Science, Freie Universität Berlin, August 2008.

[11] A. Arora, E. Ertin, R. Ramnath, M. Nesterenko, and W. Leal, "Kansei: A high-fidelity sensing testbed," IEEE Internet Computing, vol. 10, no. 2, pp. 35–47, 2006.

[12] TIK, "Notengebung bei Studien- und Diplomarbeiten." Computer Engineering and Networks Lab, ETH Zürich, Switzerland, May 1998.