FAKULTÄT FÜR !NFORMATIK

Faculty of Informatics

Configuration Change Notifications with Elektra

A notification architecture for the Elektra configuration management framework adressing emergent misbehavior

BACHELOR'S THESIS

submitted in partial fulfillment of the requirements for the degree of

Bachelor of Science

in

Software & Information Engineering

by

Thomas Wahringer
Registration Number 1025588

to the Faculty of Informatics

at the TU Wien

Advisor: Ao.Univ.Prof Dipl.-Ing. Dr.techn. Franz Puntigam
Assistance: Dr. Dipl.-Ing. Markus Raab

Vienna, 20th August, 2018

Thomas Wahringer	Franz Puntigam

Abstract

It is important for applications to be instantaneously aware of configuration changes. When multiple applications execute their configuration update logic as a result to configuration change notifications this opens the door for *emergent misbehavior* like unwanted oscillation, unwanted synchronization and phase change. Hence we investigate current approaches for detection, diagnosis and mitigation of emergent misbehavior. Informed by this investigation we propose and implement a notification feature for the configuration management framework Elektra. We present guidelines for users of the notification feature and show in which scenarios applications benefit from the use of notifications.

Contents

CHAPTER 1

Introduction

1.1 Problem Statement

In areas where configuration is frequently changed it is vital for computer applications to be instantaneously aware of *configuration changes*. Awareness of configuration changes allows applications to implement reactions to changes at run-time. We avoid outdated configuration and application restarts.

Configuration provides information about an application's environment and available resources like files, printers or database servers. Over time resources are added, change location or become unavailable. Unexpected behavior will occur if an application is not aware of these changes and tries to utilize unavailable resources.

When an application requires a restart due to configuration changes it becomes unavailable. This problem can be solved by adding redundancy to the system architecture. Using this strategy some application instances can restart while other instances keep running with the previous configuration, therefore maintaining the overall system availability. For example, multiple instances of a web service can run with a load balancer distributing incoming requests between those instances. But this strategy adds complexity and is not always feasible.

In summary, applications that are aware of configuration changes have less sources of error. They improve user experience through increased availability while adapting faster to environmental changes.

Elektra is a framework that allows applications and system operators to read, validate and write their configuration using a global key database. What Elektra currently lacks is an integrated way of notifying and updating applications when configuration changes occur. This does not only solve the problem of restarts and outdated configuration but also reduces the timespan until a new configuration is applied since customized update logic can be executed within the application.

When applications react to configuration changes made by other applications this can lead to *emergent behavior*. We speak of emergent behavior when the parts of a system are functioning as designed but an unintended, unexpected or unanticipated behavior at system level occurs. For example, take the following sequence of events:

1. application Γ changes its configuration
2. application Δ receives a notification about the change from Γ and updates its configuration

Given these two steps the sequence could be a case of *wanted* emergent behavior: Maybe application Δ keeps track of the number of global configuration changes. Now consider adding the following events to the sequence:

3. application Γ receives a notification about the change from Δ and changes its configuration
4. *continue at step 2*

The additional step causes an infinite cycle of configuration updates which introduces emergent *mis*behavior.

We need to differentiate between *wanted* emergent behavior and *unwanted* emergent misbehavior. The first can be beneficial for a system of applications while the latter can result in unwanted oscillation, livelock or unwanted synchronization[1]. Examples of emergent misbehavior in the context of configuration are frequently found in popular applications like KDE[2] or Consul[3] and embedded systems like routers[4].

For example, within the KDE desktop environment two components were working well on their own but in conjunction caused a race condition. The first component (the *X11 window system*) indicated that the configuration had changed because a new display became available. Then the second component (the *kscreen daemon*) tried find a suitable monitor configuration. Kscreen would apply the configuration to the X11 window system resulting in another configuration change notification. As a result the daemon would again search for a configuration and apply it. The workaround was to stop reacting to configuration changes until the window system had applied the new configuration.

1.2 Goals of this Thesis

The aim of this thesis is to propose and implement a *notification feature* that adds push abilities to Elektra. Applications will be notified about configuration changes and allowed to update their configuration at run-time. Notifications shall not be limited to a single *transport* technology since system operators shall be able to define which kind of transport is used. The transport message format shall be documented to allow interoperability between applications using Elektra and applications using these transports. Transport technologies like D-Bus or ZeroMQ shall be supported. These will be implemented as a proof-of-concept for the notification architecture.

We will analyze the new feature for possible sources of emergent misbehavior. In this thesis we will briefly cover emergent behavior and focus on emergent misbehavior. We will exclude intentional malicious behavior or the exploitage of emergent misbehavior. Due to the decentralized nature of Elektra applications will need to detect emergent misbehavior based on their local environment. In response to configuration change notifications applications will execute arbitrary logic, hence it is only possible to design argound those *black boxes*. The lack of a global system view (we only have a local environment) or a complete system specification (we only have black boxes) makes automated mitigation difficult. Therefore we will discuss mitigation of emergent misbehavior theoretically and present guidelines for emergent misbehavior mitigation.

The implemented notification feature will be evaluated by conducting a case study. A simple application with configuration provided by Elektra will be evaluated using the new functionality.

Based on the defined scope the following research questions need to be answered:

RQ 1 (Emergent Misbehavior) *What kind of emergent misbehavior can result from change notifications and how can it be mitigated?*

RQ 2 (Comparison) *How does the use of notifications compare to polling in an application in terms of feasibility, lines of code and CPU usage?*

CHAPTER 2

Emergent Behavior and Elektra

In this chapter we will expand on emergent behavior and examine current approaches for handling emergent behavior and emergent misbehavior in particular.

We will analyze Elektra's architecture and draft requirements for handling emergent behavior. We then review current approaches for detection, diagnosis and mitigation of emergent behavior and explain how they apply to Elektra and our notification feature.

2.1 Background

When designing a system it is desirable to use components with predictable and well-defined behavior. As a system grows larger and gets more *complex* unpredictable behavior emerges that was neither intended by the system designer nor by the designer of the components. This system behavior is called *emergent behavior* if it cannot be explained from its components but only from analysis of the whole system.

Emergent behavior is not only found in *complex systems* but also in other fields such as control systems[5]. Typical examples of emergent behavior are flocking of birds[6] and Conway's Game of Life[7]. These examples are frequently used in research for simulations[8][9][10]. In these simulations unexpected behavior emerges from a set of simple rules.

Emergent behavior can be beneficial for a system (e.g. useful cooperation in an ant colony) but it also has disadvantages. Systems that bear *unwanted* emergent behavior are difficult to manage and experience failures in the worst case. This kind of unwanted emergent behavior is called *emergent misbehavior*.

Examples of emergent misbehavior are traffic jams or the Millenium Footbridge incident in London. The day the Millenium Footbridge was opened to the public it encountered "unexpected lateral movements [...] as pedestrians crossed the bridge"[11]. As a result of these movements pedestrians had difficulties walking over the bridge. The bridge was closed down for further investigations. It was found that pedestrians had synchronized with the bridge's natural swaying frequency which further increased the amplitude of the bridge's movements. The behavior of the pedestrians was not expected and additional dampening had to be installed before the bridge was reopened.

Mogul[1] reports an example of emergent misbehavior in the context of configuration. The system in his example consists of a load balancer, two application servers and two database servers connected to each of the application servers. The load balancer distributes requests from clients to the application servers. The load balancer also detects failures of an application server when it fails to respond within a certain time limit. At first the system works as expected but suddenly the load balancer declares failure of both application servers. Due to the amount of data stored the database server's response had increased over time to a point where the application servers exceeded the load balancer's time limit. While no component had failed the whole system experienced emergent misbehavior.

Fromm[12] introduced a taxonomy of emergent behavior. It consists of types I to IV. Each type builds on the preceding type. These types describe *simple* emergent behavior, *weak* emergent behavior, emergent behavior with *multiple feedback* and *strong emergence*. Because systems of applications using Elektra have active components (applications) and feedback they are classified as at least type II according to the taxonomy.

In order to treat emergent misbehavior we need to *detect* whether an emergent behavior is present, then we need to *diagnose* it in order to differentiate whether the behavior is wanted. Finally if unwanted behavior is present, we need to apply *mitigation* to counter it.

A group of applications using Elektra potentially constitutes a large *complex system* with emergent behavior. This complex system consists of applications as components which are connected by configuration storages. With the addition of the *notification feature* components are additionally connected via *notification channels*. This notification architecture also allows applications to run custom *update logic* on configuration changes. This update logic enables us to circumvent application restarts thereby decreasing adaption time on configuration changes. Then again, it also increases the possibility for emergent behavior by allowing applications to directly react to changes made by other applications.

The composition of applications using Elektra cannot be anticipated by neither Elektra's developers nor application developers. The set of active components is constantly changing as applications are started and stopped. This implies that a system's emergent behavior is also constantly changing. As a result emergent behavior can only be treated when it occurs *live* at run-time. In this case interventions by humans are not feasible which makes *automatic* treatment necessary.

Elektra is designed not to have a single point of failure. Therefore there is no *central* entity (i.e. background process or daemon) that controls access to the key database.

Each application uses functions from the Elektra library and stores in its memory a copy of the part of the global key database it is working with. This requires *decentralized* treatment of emergent behavior at the component level.

In summary Elektra requires *decentralized automatic live* detection, diagnosis and mitigation of emergent behavior.

2.2 Detection

Detection is responsible for the decision whether emergent behavior is present in a system. In order to be able to detect emergent behavior we need to detect if a system deviates from normal system behavior. Depending on the approach we also need to define normal system behavior. Existing approaches for detection of emergent behavior can be categorized into *formal*, *event-based* and *variable-based* approaches. We will give a short overview of each category and discuss to which extent they are applicable to our system of applications using Elektra.

Formal approaches are based on formal languages. Most formal approaches use *grammars*[13] to deduce emergent properties from the difference between states of the whole system and superimpositions of the system's component states. In order to create grammars for these approaches it is required to know all possible component behaviors. Due to the number of combinations approaches in this category suffer from state-space explosion as systems get more complex. These approaches do not require prior observation of the system in order to find possible candidates for emergent behavior properties because they are deduced from the grammars.

There are also approaches using *logic formulae*. A decentralized approach detects emergent behavior by evaluating these logic formulae in a network of components[14]. This approach requires the description the expected emergent behavior in formulae.

Model-based approaches which also fall into this category rely on domain knowledge to construct the required models. For example, an approach uses message sequence charts and *semantic causality* to determine what message caused another message[15]. According to this approach emergent behavior arises when the resulting finite state machines have *identical states* which can lead to ignoring messages.

For our system it would be required to know all possible component behavior in order to be able to construct grammars. This makes grammar-based approaches unsuitable for our purposes since the set of executed applications is constantly changing. There is no way to know every possible behavior at design-time. Formulae-based approaches contain the exact conditions for specific types of emergent behavior encoded in their logic formulae. This limits the the type of detectable types of emergent behavior. Because we will only provide the framework for sending and receiving notifications a part of the model specification is filled in later by applications. Therefore required domain knowledge is missing for model-based approaches.

Variable-based approaches process component *variables* or derived metrics that describe component state or relations between components (i.e. interactions). Emergent behavior is detected from variable changes using statistical analysis, graph analysis or other techniques. For example, an approach detects emergent behavior by measuring entropy at the start and end of a simulation run[16]. Emergent behavior is present if entropy has decreased over time. Typical variable-based approaches require domain knowledge to define which variables to observe.

These approaches depend on the selection of variables for observation in order to allow detection of emergent behavior. Since a system of applications using Elektra inherently has a constantly changing system behavior, emergent behavior is likely also dependent on the set of involved components and is therefore, constantly changing too. This reveals that we cannot sufficiently select variables in advance that ensure that emergent behavior can be detected. However, an approach by O'Toole, Nallur, and Clarke[17] performs automatic selection of variables at run-time using statistical methods.

Event-based approaches are based on *single and complex events*[18]. Single events capture a state change of multiple variables over a duration of time. Complex events are composed from single and other complex events. Emergent behavior is detected from the presence of selected complex events that were defined using domain knowledge. In event-based approaches the *sequence of events* describes what lead to an instance of emergent behavior.

Event-based approaches require domain knowledge for definition of complex events that allow detection of emergent behavior. We may be able to define such events in advance informed by yet to be discovered typical patterns of emergent *mis*behavior in systems of applications using Elektra. These events will be limited in which emergent behavior can be detected since their definitions are based on assumptions about specific patterns and they are focused on emergent misbehavior. Despite this focus on emergent misbehavior *diagnosis* is still relevant to detect false positives.

In the Section 2.1 we have established that a system of applications using Elektra requires decentralized automatic live detection of emergent behavior. Additionally the set of components and therefore, behavior in our system is constantly changing.

Event-based approaches require prior definition of events and are only able to detect emergent behavior encoded into these event definitions. The formulae based approach has the same limitation. Grammar based approaches require knowledge about all states. Even if it were possible to enumerate all states from applications the grammars would have to be rebuilt whenever the set of running applications changes. From the previous categories variable-based approaches are best suited for our purposes.

There exist many variable-based approaches for detection of emergent behavior[9][19] but as Elektra requires decentralized detection it is necessary to detect emergent behavior *locally* at the component level.

Emergent behavior generally occurs when actions at the component level result in a change of system level behavior. This is called *upward causation* and is present starting with type I of Fromm's taxonomy. In systems with type II emergent behavior components are also affected by the system behavior in the opposite direction through *top-down feedback*. For example, in a traffic jam the volume of cars (or components) creates an emergent behavior (congestion), but cars are also affected with reduced speed and choice of route by the emergent behavior. This concept is called *downward causation*. As we have established, our system of applications using Elektra is classified as type II. This allows us to argue that in our system emergent behavior is detectable locally at the component level and that therefore, detection can be performed decentralized.

Downward causation is used in the decentralized variable-based approach by O'Toole, Nallur, and Clarke[17]. This approach performs automatic selection of variables using statistical tools. These variables are fed into a change detection unit which uses a sliding window and cumulative sums to dectect change points in the selected variables' time-series. Change points indicate that emergent behavior *may* be present. To reduce false positives a collaboration unit compares local results with those of randomly chosen neighbors. Since this approach can detect emergent behavior automatically in a live system it satisfies all our requirements for emergent behavior detection in our context.

2.3 Diagnosis

Once emergent behavior is detected it is necessary to perform *diagnosis* on the behavior in order to decide whether it is *unwantend* emergent misbehavior or *wanted* emergent behavior. While we want to stop unwanted behavior we do not want to stop behavior like cooperation between applications (e.g. applications sharing configuration settings). The decision between unwanted or wanted behavior is subjective and dependent on the desired goals for a system.

For many existing detection approaches diagnosis is out of scope. These approaches perform detection of emergent behavior in agent-based simulations. When a simulation shows emergent behavior the system designer decides whether the behavior is unwanted and alters the agent's parameters, rules or program and reruns the simulation until the unwanted behavior is mitigated or does not occur anymore[8][9][20]. Due to our requirement for automatic detection at run-time this design-time simulation based process is not applicable.

Some approaches do not separate between diagnosis and detection process[21][22]. An approach diagnoses emergent behavior by comparing an interaction graph with a reference graph[21]. Unfortunately this approach is not decentralized and requires domain knowledge for constructing the reference graph and classifying it as wanted or unwanted behavior.

For variable-based detection approaches defining goals is natural. These goals define thresholds for variables like “throughput” and ensure that a system is operational. If the goals are violated *and* emergent behavior is detected the current system behavior is unwanted.

An approach by Khan et al.[22] combines detection and diagnosis. It uses performance goals to add additional information the diagnosis process. These goals contain metrics like “average response time” and “response throughput” with corresponding thresholds. A system operator can supply additional information by marking events with different colors. Depending on the type of symptom the green category is used for events that counter the symptom and red reversely. For example, if high energy consumption is diagnosed, green is used for events that decrease consumption (e.g. shutdown of a virtual machine) and red is used for events that increase consumption. Because this approach performs diagnosis on a complete view of the system it cannot be performed decentralized.

Currently there exists no approach for diagnosis that fullfills our requirements for automatic decentralized live diagnosis of emergent behavior. We have introduced and discussed multiple approaches with promising ideas but we had to reject them since they did not meet our requirements.

2.4 Mitigation

Once emergent behavior has been detected in our system and eventually diagnosed as unwanted it is necessary to put countermeasures into action. Applications that detect emergent behavior could be simply cut off the system. But the emergent behavior might be so ubiquitous that is affects all applications. We also could have incorrectly diagnosed a cooperative emergent behavior as unwanted. Therefore, it is better to *mitigate* emergent behavior which keeps a system operational.

Until now we have looked at the results of emergent behavior. We started with unspecified behavior and tried to detect and diagnose it automatically. For mitigation it is necessary to look at the circumstances that encourage emergent behavior, understand them and the resulting symptoms.

Mogul[1] has suggested to create a typology of *symptoms* and *causes* for emergent misbehavior. Of these causes a system of applications using Elektra has *decentralized control* and *massive scale*. However, the presence of a cause does not directly lead to emergent behavior. These causes are merely causal factors since they contribute to or encourage emergent behavior.

An evaluation of our system shows that it can exhibit the following symptoms from the aforementioned typology:

Symptom	**Synchronization**	**Oscillation**
Suggested actions	Over design capacity; induce randomness; introduce perturbations	Decoupling; induce randomness

Table 2.1: Suggested mitigation actions for emergent behavior[23]

- *Synchronization* occurs due to the shared notification medium. Multiple applications receive a notification at the same time and execute their configuration update logic. In turn shared resources like hard disk or CPU become overutilized.

- *Oscillation* occurs when applications are reacting to configuration changes by other applications.

- In the worst case oscillation results in *livelock* when the frequency of configuration updates becomes so high that applications are only executing configuration update logic.

- *Phase change* is a sudden change of the system behavior in reaction to a minor change (e.g. incremental change of a configuration setting). While phase change is possible, it stems from application logic and is therefore, out of Elektra's scope.

Reid and Rhodes[23] are building on Mogul's idea of a typology for emergent misbehavior. In their work they have suggested to build a set of design patterns that help to identify and address both emergent misbehavior and emergent behavior. While their set of design patterns is intended to be completed by follow-up research they have also linked selected symptoms to according actions for mitigation and encouragement of emergent behavior.

As we can see in Table 2.1 inducing randomness is a common denominator of both symptoms. In our system livelock is a special case of oscillation and phase change. Therefore livelock is out of scope and we can conclude that inducing randomness is a viable action for mitigating emergent misbehavior.

To mitigate emergent behavior in our system of applications using Elektra, upon receiving a notification we recommend to randomly wait before an update is propagated to the application. This will mitigate both synchronization and oscillation. Additionally decoupling for oscillation is introduced by rejecting identical notifications during the time delay: Notifications about configuration changes of the same key that are received during the random time delay get rejected and do not start a new random delay. However, notifications about changes to other keys start a new random delay. We will use a variant of this approach later in the case study in Chapter 4.

2.5 Summary and Discussion

We have evaluated different approaches for detection, diagnosis and mitigation of emergent behavior in the context of a system of applications using Elektra:

Detection: We have found that the variable-based detection approach by O'Toole, Nallur, and Clarke[17] did meet our requirements.

Diagnosis: While we could not find a suitable approach for diagnosis we were able to determine that goals aid the decision whether emergent behavior is wanted or unwanted. The diagnosis approach by Khan et al.[22] could be decentralized by constructing a partial view from collaboration with randomly chosen neighbors. The non-automated part of coloring is out of scope for this thesis since it informs analysis of the cause of a goal violation. In this thesis we are only interested in mitigation of emergent misbehavior (RQ 1) and not in finding its cause. Validation of the outlined approach is up for future work on this topic.

Mitigation: Based on symptoms of emergent behavior and a evaluation of our system of applications using Elektra we have found countermeasures for mitigation of emergent misbehavior.

Ultimately automatic live detection, diagnosis and mitigation of emergent misbehavior can only lower the impact of faulty algorithms, applications or a combination of applications, not repair them. A system that exhibits emergent misbehavior is inherently faulty and remains so even if mitigation is applied.

Although detection, diagnosis and mitigation are constrained by Elektra's position as a library used by applications it is necessary to address emergent misbehavior close to its possible origin. From an engineering perspective it is better to address a problem at the earliest possible stage in the process to prevent increased damage or cost at later stages. At every stage from Elektra's design, developers integrating Elektra into their applications, system operators installing and configuring those applications and finally to users experiencing emergent behavior, involved roles need to be aware of emergent misbehavior in order to minimize its impact.

The custom update logic implemented in applications is a factor for emergent misbehavior in our system. It is responsible for potential phase changes and for the likelihood of synchronization, oscillation and livelocks. Therefore design guidelines for developers for preventing emergent misbehavior are necessary.

Separate plugins will log application interactions with the key database to help developers analyze possible problems. No automatic detection or mitigation is performed.

We recall our first research question RQ 1:

RQ 1 (Emergent Misbehavior) *What kind of emergent misbehavior can result from change notifications and how can it be mitigated?*

We have established that a system of applications using Elektra requires decentralized automatic live treatment of emergent behavior. Since the set of applications is constantly changing, the behavior of the system also changes. Therefore its emergent behavior (including emergent *mis*behavior) cannot be predicted in advance. Instead we have outlined requirements and possible approaches for detection and diagnosis of emergent behavior. Furthermore we have examined possible symptoms of emergent misbehavior in our system (synchronization, oscillation, livelock and phase change) and introduced possible mitigations for these symptoms where necessary (induce randomness, decoupling).

CHAPTER 3

Implementation

In the introduction we defined the requirements for the notification feature: Applications should be notified about configuration changes and be able to update their configuration at run-time. Additionally notifications should not be limited to a single technology for transporting notifications.

In the previous chapter we have found that in order to treat possible symptoms of emergent misbehavior in a system of applications using the notification feature we have to induce randomness and introduce decoupling when delivering notifications (Table 2.1 on page 11).

Building on these requirements and findings we will now present the architecture of notifications for applications using Elektra. We will give a short overview and provide details for each involved component.

3.1 Background

The Elektra framework consists of libraries and multiple tools for accessing the global key database (KDB). In this database a configuration *setting* consists of a *key* and its *value*. The key database is organized hierarchically so that any application can store its settings in it. It is possible to *mount* configuration files in a similar fashion to devices in a UNIX file system. Elektra has a minimal core and most of its functionality is implemented by *plugins*. Elektra's plugins implement different functions like storage, validation and filtering. Before this work over 80 plugins existed of which one implements notification functionality. This plugins sends notifications via D-Bus (a message bus system for Linux desktop systems) when a setting is changed. In order to use a plugin with Elektra it is required to either mount it with a configuration file or mount it *globally*. When a plugin is mounted globally all configuration settings are passed through it.

Currently, to get configuration updates, applications have to repeatedly read from the key database (also called *polling*) or watch for notifications on D-Bus. As stated in the introduction we want applications using Elektra to be notified about configuration changes and allow them to update their configuration at run-time. Besides sending notifications this also requires receiving them. Applications could directly use plugins but this would couple them to specific plugins. Instead we will create a notification application programming interface (API) which allows applications to receive configuration updates in a more integrated and user-friendly way. By using an API we also decouple applications from the plugins that transport the notifications: Instead of D-Bus we can use other techniques for sending and receiving notifications.

3.1.1 Asynchronous Processing

In order to be able to receive notifications *asynchronous processing* is required. Without asynchronicity programs would stop (or *block*) until a new notification arrives. A program is a series of *instructions*. The *control flow* dictates in which order instructions are executed. When a notification is available the normal control flow needs to be interrupted to process the notification. In order to detect available notifications it is also required to know when an underlying transport has data available. For example, a network service like D-Bus uses *sockets* to transport its data. To detect when data is available applications normally use I/O functions like `select` or `poll`. However, these functions block the control flow until data is available. To circumvent blocking applications can create *processes* or *threads* and handle notification processing there. Processes and threads have their own separate control flow and therefore do not interfere with the normal control flow. However, applications need to synchronize with these processes and threads which is error-prone and hard to maintain.

I/O functions like `select` allow us to pass a *timeout* after which the function returns if there is no data available. This timeout can be arbitrarily short. In this case the function will return immediately - with or without data. Embedded in a loop these functions allow an application to do other tasks and check again later if there is data available. This is the basic idea of an *event loop*. In event loops time-consuming tasks are divided into slices and executed among other slices. Every iteration the event loop checks if other *events* (e.g. “socket has data”) are available.

Event loops are often used in graphical user interface (GUI) applications (i.e. Windows, Gnome or KDE) but are also used for network services (i.e. Netty[1] or Node.js[2]). Event loops enable separation of application specific code from the event loop implementation. This also enables libraries like Elektra to integrate into event loops. There are multiple event loop libraries available that abstract away system and I/O function calls.

[1] https://netty.io/, accessed August 2018

[2] https://nodejs.org/, accessed August 2018

3.2 Architecture

Elektra's notification feature consists of a notification library, plugins for handling the transport of notifications and a solution for allowing asynchronous processing called *I/O binding* (see Figure 3.1).

The *notification library* implements the API for receiving and handling notifications. Applications use this API to subscribe to changes of settings. Internally the library uses a plugin to implement the functionality. This enables the notification library to feed local changes back into an application.

Notifications are *transported* using different plugins. Each type of transport typically has two *transport plugins*: one for sending and one for receiving notifications. The system operator decides which plugins are used and mounts them globally.

In order to receive notifications, transport plugins require asynchronous processing. While we want to use different transport plugins we also need to support integrating with different event loop libraries. The solution is an abstraction layer for asynchronous processing called I/O binding. Components that are independent from the event loop library (e.g. transport plugins) use the *I/O interface* implemented by the I/O binding which implements integration with a specific event loop library. This allows asynchronous notification processing by compatible plugins while being able to integrate into virtually any event loop.

Separate plugins are responsible for logging when applications get and set configuration settings. Since these plugins get mounted globally by system operators when required they are not coupled to other components in the architecture. The resulting logs are later analyzed for emergent misbehavior by either operators or application developers.

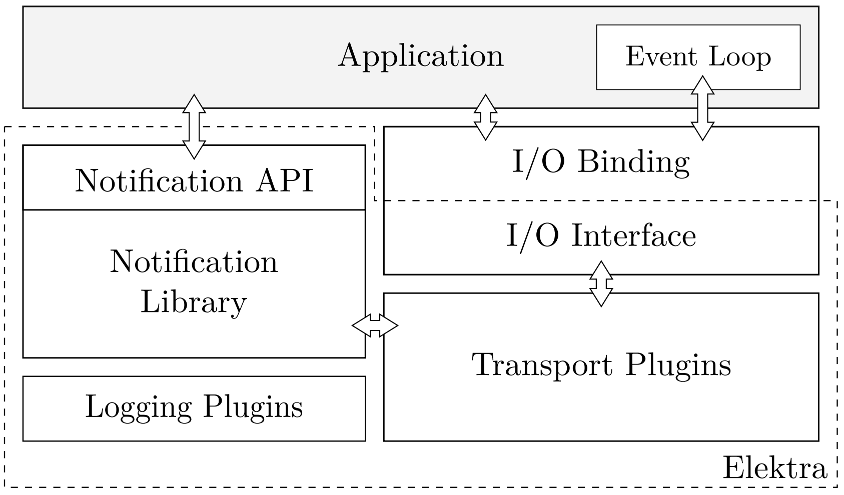

Figure 3.1: Notification architecture

From an application developer perspective an application will initialize its event loop, create an I/O binding and pass it to Elektra. Afterwards applications will initialize the notification feature using the notification API. The notification library mounts its internal plugin globally and initializes mounted transport plugins. Then the application will subscribe to changes of configuration settings it uses and continue with normal application logic. We will explain how notifications are generated and processed in Section 3.5.

3.3 Notification API

Applications will use the notification API to implement notification features. The notification library which implements this API provides the `elektraNotification-Register` family of functions to let applications express interest in notifications about changes to configuration settings.

The API will offer the ability to register application variables for changes to settings for all of Elektra's supported basic CORBA types (e.g. boolean, short, long or float). Registered variables will be updated automatically when a setting is changed. When more complex logic is required after a configuration change (e.g. rebuilding data structures) applications will be able provide a callback function that is called whenever a setting has changed.

3.4 Asynchronous Processing Abstraction

Transport plugins will use different libraries to send and receive notifications. In order to do so without blocking the application, they use an abstraction layer for asynchronous processing called I/O binding.

The I/O binding integrates into an application's event loop which allows other processing to continue while waiting for long-term tasks like network or file operations to finish. Elektra will include bindings for common libraries like glib or libuv, but application developers can implement their own bindings. Transport plugins will be implemented against the I/O interface in order to be able to use different I/O bindings.

Use of an I/O binding is not necessarily bound to transport plugins or Elektra. It can be used by other parts of Elektra to perform asynchronous operations. It can also be used in applications to integrate libraries that require asynchronous event processing by using a generic interface. This saves developers from the repetitious task of integrating a library with a specific asynchronous event processing library.

The I/O interface is a data structure that contains pointers to the functions for managing *file descriptors*, *timers* and *idle* taks. The I/O interface is allocated and populated by the I/O binding's initialization function and assigned to a KDB instance. This architecture enables multiple and different bindings to be used in an application.

Timers allow us to schedule operations at regular time intervals. Idle operations allow execution of long-time low-priority tasks when no other high-priority events require processing. On POSIX compatible systems a file descriptor is the most common basic building block for notification about available data and asynchronous event processing. It is used by various functions like `select` on Linux and `kqueue` on BSD. Other systems are out of scope for this thesis.

3.4.1 Testing

In order to ensure that I/O bindings are interchangable an extensive test suite is provided which ensures that operation characteristics are consistent.

Application developers can also test their custom bindings against this test suite.

3.5 Transport Plugins

A system operator defines which transports are used for notifications by mounting the required plugins globally. Since all configuration settings are passed through the plugins every change is detected.

For each type of transport there are separate plugins for sending and receiving notifications. Receiving plugins perform no operations unless asynchronous processing by an I/O binding is provided and the notification feature was initialized by an application. Sending plugins will work without the notification feature initialized thereby enabling every application using Elektra to send notifications without source-code changes.

A notification is generated when a sending transport plugin detects a change to configuration settings. These sending plugins can use the `postcommit` plugin hook which is triggered after a change has been applied successfully to the key database to inform other applications about changes. A sending transport plugin is not required when other sources for triggering a notification are used (e.g. triggers in database).

For every transaction the name of the topmost changed configuration setting is included in the notification which is sent over a communication channel. Instead of sending notifications for every individual changed setting we take advantage of the hierarchial structure of the key database. This reduces the amount of notifications while maintaining the ability to notify which parts of the key database have changed.

The receiving transport plugin of another application using Elektra subscribes to changes of settings on the same communication channel as the sending transport plugin. When a notification is received the changed setting is forwarded according to the application's registration to the notification library which loads the changed settings from the key database.

This implicitly triggers the internal plugin of the notification library. Depending on the registration done by the application either a variable is automatically updated or a callback supplied by the application for the changed setting is called.

3.5.1 Characterization of Transports

A notification transport needs to be capable of delivering notifications to from a single sender to multiple receivers. A sender is unware of the receivers and uses a service to deliver notifications. Reversely receivers are unaware of the senders and only connected to them through a service. Furthermore a receiver shall not be blocked while waiting for the next notification.

These requirements are fulfilled by publish/subscribe systems. Within such systems publishers *publish* information about *events* to an *event service*[24]. Subscribers *subscribe* with the event service and receive *notifications* about events. Publish/subscribe systems provide *time decoupling*, which means that the publisher of an event and subscribers receiving the event must not be connnected to the event service at the same time. Both publishers and subscribers do not interact directly with each other but only through the event service, which is called *space decoupling.* Furthermore publishers are not blocked during sending until all subscribers have received notifications and conversely subscribers are not blocked until the next notification arrives, which results in *synchronization decoupling.* Elektra can take advantage of decoupling in space and synchronization within publish/subscribe systems.

3.5.2 D-Bus Transport Plugins

The existing D-Bus plugin was transformed into a transport plugin for sending notifications.

For receiving notifications via D-Bus a new plugin was implemented. In order to receive D-Bus *signals* a "match-rule" is sent to the D-Bus daemon while incoming signals are processed.

3.5.3 ZeroMQ Transport Plugins

Proof-of-concept transport plugins using ZeroMQ's publish/subscribe sockets were created.

ZeroMQ provides a file descriptor for signaling which is implemented with *edge-triggererd* notifications. As a result the descriptor signals only once when notifications are available and does not signal again until the message queue is emptied.

A naïve solution is to consume all notifications immediately but this potentially time-consuming operation would block the event loop if notifications were received continuously. To combat this issue, idle operations were be used to read remaining messages without interfering with other operations.

Since ZeroMQ did not provide publish/subscribe with multiple subscribers and publishers out of the box a intermediary message hub was implemented.

3.5.4 Comparison of Transports

D-Bus is the de-facto standard for Linux desktop IPC and mainly targets single hosts. D-Bus is available for many desktop systems or can potentially be used on them. It is available for Linux, FreeBSD and ports for MacOS are available.

Although ZeroMQ can be used to spread notifications over a network, its local inter-process communication (IPC) transport can be used as an alternative to D-Bus on POSIX compatible operating systems (e.g. for embedded systems).

3.5.5 Configuration

Application developers can only specify whether or not they want to receive and handle notifications, but not which transport plugin is used.

System operators can mount the desired transport plugins and configure them (e.g. set channel, host, port and credentials) globally.

System operators need to mount receiving and sending plugins to use a transport, as shown in section 4.2.7 of our next chapter, “Case Study”.

CHAPTER 4

Case Study

In this chapter we will present our findings from a case study conducted to show the feasibility and practicability of the new notification feature presented in the previous chapter. We have adapted an existing application to use the notification API.

After replicating the required steps and highlighting both benefits and possible pitfalls we will compare the new application version with its original version. Finally we will present guidelines designed to handle emergent misbehavior using the notification API and discuss our findings from the case study.

4.1 Application: Humiditycontrol

The application we adapted is called “humiditycontrol”. This application reads data from humidity and temperature sensors inside and outside a building and operates ventilation based on several target values like heating cost factor and maximum desired humidity. This application is targeted for use in an embedded system therefore resource-efficient operation is desired.

The application uses a library called “libautomation”[1] which aims at facilitating creation of industrial and home automation applications. It supports centralized data acquisition and distributes data source readings among applications using shared memory. The library uses “libev” a library for asynchronous processing with event loops.

Before the case study the application “humiditycontrol” read its configuration containing data source paths and target values from a file. In this file a configuration setting consisted of a key and a value separated by a horizontal tab (`\t`).

[1] https://repo.or.cz/libautomation.git, accessed August 2018

4.2 Implementation

For this case study we will change the application to only use Elektra's new notification API for reading target values from configuration.

To this end we need to perform the following steps:

1. Link the application against required libraries
2. Add the required header files for Elektra and its notification API
3. Open Elektra's Key Database (KDB)
4. Create and set an I/O binding
5. Initialize the notification API
6. Register variables holding target values
7. Verify that target values are updated at run-time

4.2.1 Linking and Headers

First we have to link the application against all required libraries. The notification and I/O API are implemented by the libraries *elektra-notification* and *elektra-io*. Since we will integrate into the main loop of the application we will use the I/O binding for "libev" which is implemented by the *elektra-io-ev* library.

The next step is to include the required header files.

```c
#include <elektra/kdb.h>
#include <elektra/kdbio/ev.h>
#include <elektra/kdbnotification.h>
```

Listing 4.1: Required header files

In this step we experienced a syntax error caused by a name collision of `KEY_MODE` used as argument for Elektra's `keyNew()` function with the definition of a keyboard input in the "linux/input-event-codes.h" header file. A quick workaround was to rearrange the header files so that Elektra's header files were included first.

We also found that `KEY_END` was used by both Elektra and the "linux/input-event-codes.h" header file. Since `KEY_END` was not already used in the source code we could use Elektra's definition.

4.2.2 Opening Elektra's Key Database

To start reading and storing configuration settings from the global key database we need to create an instance of KDB by calling `kdbOpen()`. Configuration settings within KDB are represented by the `Key` type. The `kdbOpen()` function uses such a `Key` to store additional diagnostic information like warnings and errors.

Usually a `Key` containing the name of the root of the application configuration settings is used. This `Key` is used in the next steps to obtain these settings.

Since Elektra's key database is used by multiple applications a naming convention applies for storing and retrieving application specific configuration settings. We will use the name `"/sw/libautomation/humiditycontrol/#0/current"` for this application. Per convention this name consists of several parts separated by `"/"` to indicate a new hierarchy level:

1. `sw` stands for software
2. `libautomation` is the URL or organization name
3. `humiditycontrol` is the application name
4. `#0` is the major version of the configuration settings. If an application introduces incompatible changes to its configuration settings (e.g. the type of a value is changed from string to numeric) this number is increased.
5. `current` is reserved for profiles. This allows applications to have multiple user profiles.

On the top-level of the hierarchy there are so called *namespaces*. Namespaces have different meanings and write semantics. For example, the `system` namespace is only writable by system operators and meant to store system default settings. The `user` namespace is user specific and writable by users.

A powerful feature of Elektra is *cascading* which is used when a setting is accessed using `"/"` at the beginning of a `Key` name. When a setting is accessed using cascading Elektra's namespaces are searched for the setting in a specific order. For example, the `user` namespace is searched before the `system` namespace. This allows users to override system defaults.

The code shown in Listing 4.2 opens the key database. The `atm_fail()` function prints an error and exits the application. All functions in this chapter starting with `"atm_"` are part of the "libautomation" library.

```c
int main(int argc, char **argv){
  // Variable declarations, libautomation initialization,
   ↪  etc.
  // ...

  // Open KDB
  parentKey = keyNew
   ↪  ("/sw/libautomation/humiditycontrol/#0/current",
   ↪  KEY_END);
  kdb = kdbOpen (parentKey);
  if (kdb == NULL) {
    atm_fail ("kdbOpen failed");
  }
```

Listing 4.2: Opening KDB

4.2.3 I/O Bindings

Now that we have created a KDB instance we have to create an I/O binding. As explained in Section 3.2 on page 17 an I/O binding allows Elektra to integrate into an event loop to perform asynchronous processing.

```c
// Open KDB
// ...

// Init I/O bindings
ElektraIoInterface * binding = elektraIoEvNew
 ↪  (EV_DEFAULT);
elektraIoSetBinding (kdb, binding);
```

Listing 4.3: Using an I/O binding

In Listing 4.3 the function `elektraIoEvNew()` creates an I/O binding for the "libev" default main loop which is obtained using `EV_DEFAULT`. The function `elektraIoSet-Binding()` sets the I/O binding for the application's connection to the key database. Only the two lines above were necessary to make Elektra integrate into the application's main event loop.

4.2.4 Initialize Notifications

The next step is to initialize the notification feature. In Listing 4.4 a call to `elektraN-otificationOpen()` mounts the internal notification plugin globally for the process.

This allows the API to detect changes to configuration settings and update registered variables. If present receiving transport plugins are also intialized by this function call. This allows plugins to connect to a service and start receiving notifications.

```c
// Init I/O bindings
// ...

// Initialize Notifications
if (!elektraNotificationOpen (kdb)) {
  atm_fail ("could not open notification");
}
```

Listing 4.4: Initialization of the notification API

4.2.5 Register Variables

We will now register the target values from our application for automatic updates.

When a configuration setting holding a target value is changed the according variable in the running application is automatically updated. Since the application runs its control routine at regular intervals automatic updates are well suited for this type of application: After an update the control routine bases its next calculation on the new values.

```c
1  while(getline (&line, &len, f) > 0){
2    if (line[0] == '#')
3    continue;
4
5    first = strtok (line, " \t");
6    rest = strtok (NULL, "\n");
7
8    if (!strcmp (first, "out_temp"))
9    tempo = atm_ds (rest, first);
10   // ...
11   else if (!strcmp (first, "interval"))
12   interval = atm_read_float_or_fail (rest, first);
```

Listing 4.5: Configuration parsing before the case study

Before the application used Elektra the code in Listing 4.5 was used to parse configuration settings. The code reads the configuration file line by line and uses the `strtok()` string processing function to extract the name of the setting (stored in the variable `first`) and the remainder of the line (`rest`). In lines 8 and 9 the setting for `out_temp` which is a data source is read and applied. In lines 11 and 12 the `interval` setting is read. The name of the function `atm_read_float_or_fail()` is self-explanatory: It tries to convert the content of `rest` to float. If this operations fails the applications exits.

Using Elektra's notification API in Listing 4.6 variables are registered with `elektraNotificationRegisterDouble()`. This function takes a handle to the KDB instance, a `Key` holding the name of the settings (in this case "`/sw/libautomation/humiditycontrol/\#0/current/interval`") and the address of the variable that should be updated on changes.

The semantics for all `elektraNotificationRegister()` functions are: *Iff* a configuration setting is present and it has a valid value (e.g. correct format) then the registered variable is updated.

```
keyAddBaseName (registerKey, "interval");
if (!elektraNotificationRegisterDouble (kdb, registerKey,
  ↪  &interval)) {
  atm_fail ("could not register interval");
}
```

Listing 4.6: Registration of variables

The "interval" setting in the application was declared as `ev_timestamp`. We assumed `ev_timestamp` was defined as `float` type (floating point with single precision). On compilation it turned out that it was defined as `double` (floating point with double precision). Since all functions for registering variables are typed this mistake was caught early.

4.2.6 Polling

The next step is to retrieve configuration settings from Elektra's key database. We already have initialized KDB, an I/O binding and the notification feature. By calling `kdbGet()` Elektra will automatically update registered variables if matching configuration settings are present. Since in this application we only use registered variables we can discard the configuration immediately using `ksDel()`.

```
// Update elektra configuration manually
KeySet * config = ksNew (0, KS_END);
kdbGet (kdb, config, parentKey);
ksDel (config);
```

Listing 4.7: Retrieving configuration settings

As mentioned before the application performs its control routine at regular intervals. We also added the code from Listing 4.7 to the control routine. This polls Elektra's key database for configuration updates. No further actions are required to keep registered variables synchronized with the key database.

4.2.7 Using Transport Plugins

Performing manual updates through polling is not ideal. We will now remove the additional code shown in Listing 4.7 from the control routine and use transport plugins to send and receive notifications on configuration changes.

Since the application is targeted at embedded systems we will use the ZeroMQ transport plugins called "zeromqsend" and "zeromqrecv". ZeroMQ is well suited for embedded systems due to its small memory footprint, small library size and small number of dependencies.

Sockets provide an 1-to-n mapping since their communication scheme requires a well known *endpoint* (e.g. an IP address and port): One socket *binds* to an endpoint while several sockets *connect* to this endpoint. In case of a publish/subscribe system this enables one publisher to talk to multiple subscribers or one subscriber have multiple publishers.

This does not satisfy our requirements for a notification transport (see Section 3.5.1). Therefore a *hub* is required that provides well known endpoints for both publishers and subscribers to connect to. This hub distributes notifications among participating plugins embedded into applications using Elektra.

The "zeromqsend" plugin detects changes to configuration settings and sends notifications over ZeroMQ's PUB socket. The hub receives notifications and forwards them to connected subscribers. The "zeromqrecv" plugin receives notifications from the hub using a ZeroMQ SUB socket and forwards notifications to the notification library.

Using different transport plugins requires no modification of the application's source-code. The first line from Listing 4.8 is typically executed when the system boots. The second line which mounts the transport plugins is typically executed when the application is installed.

```
kdb run-hub-zeromq
kdb gmount zeromqsend zeromqrecv
```

Listing 4.8: Global mounting and starting the hub

For the ZeroMQ transport we also have to ensure that the hub is running.

4.2.8 Summary

The transformation from the original application to the current version[2] using Elektra's key database with notifications took about 50 minutes. During the transformation 15 lines of code from the configuration parser (Listing 4.5) were made obsolete and 44 lines were added. For lines of code we use the source lines of code (SLOC) metric which excludes comment lines.

After this transformation the application is able to implement changes to its configuration at run-time.

Besides the fixed amount of lines required for initialization, the registration of a `Key` takes four lines of code using the notification API. Previously parsing a configuration setting required two lines of code.

4.3 Callbacks

Registering a variable is suitable for applications where the variable's value is simply displayed or used repeatedly (e.g. by a timer or in a loop). If an initialization code needs to be redone after configuration changes (e.g. a value sets the number of worker threads) updating a registered variable does not suffice. For these situations a callback should be used.

```c
static void initKdb (void) {
  if (kdb != NULL) {
    // Cleanup kdb
    elektraNotificationClose (kdb);
    kdbClose (kdb, parentKey);
    elektraIoBindingCleanup (binding);
    keyDel (parentKey);
  }

  // Initialization code
  // ...
}
```

Listing 4.9: Function with initialization logic for Elektra

We now add a callback for the configuration setting `"system/elektra"`. Below this setting Elektra stores internal information about the key database like mount points and globally mounted plugins (both described in Section 3.1 on page 15). A callback for this

[2]https://repo.or.cz/libautomation/elektra-notification.git commit `dce371f` in branch "thesis", accessed August 2018

setting allows the application to react to changes to the key database configuration and reload accordingly.

In order to reload the key database we moved the initialization code for Elektra into a separate function called `initKdb()` (Listing 4.9) and added code for closing the key database and cleanup. The remainder of the function is initialization code as shown in Sections 4.2.2 to 4.2.5.

```c
keySetName(registerKey, "system/elektra");
if (!elektraNotificationRegisterCallbackSameOrBelow(kdb,
 ↪ registerKey, elektraChangedCallback, NULL)) {
  atm_fail("could not register for changes to Elektra's
   ↪ configuration");
}
```

Listing 4.10: Register for changes to Elektra's configuration

In Listing 4.10 we register a callback for changes below `"system/elektra"` by calling the `elektraNotificationRegisterCallbackSameOrBelow()` function after registering variables.

Initially we used the function `initKdb()` directly as callback. This did result in an application crash since after the callback had closed the key database connection the logic inside the notification API tried to access already removed data structures. The solution was not to use `initKdb()` as callback but set a *flag* indicating that the key database should be reloaded and wait until the control flow is back in the main event loop and *then* reload.

While verifying that our callback worked as desired we discovered a bug that resulted in application crashes when using the “zeromqrecv” plugin. As a workaround we continued with the “dbus” transport plugins and were able to mount a configuration file with a different setting for the “highhumidity” target value. Upon mounting the application would reinitialize KDB and use the new value from the mounted file through the registered variable. When unmounting the configuration file the application would revert to the old value again.

Once the bug that resulted in an application crash with the “zeromqrecv” plugin was fixed we were able to complete a roundtrip of notification transports: After the application was started with the ZeroMQ transport plugins mounted globally, D-Bus transport plugins were added. After we verified that the application was now able to receive notifications over both transports we removed the ZeroMQ transport plugins. Upon verification that the application was able to receive notifications over only the D-Bus transport we added the ZeroMQ transport plugins again and removed the D-Bus ones. Now the application was back to the original set of transport plugins.

4.3.1 Guidelines for Using Notifications and Callbacks

In Chapter 2 we have found that in our system of applications using the notification feature we have to induce randomness and introduce decoupling when delivering notifications. Building on these findings and lessons learned in this chapter we will now present guidelines for preventing emergent misbehavior when using the notification API and callbacks in particular.

We recall our first research question RQ 1:

RQ 1 (Emergent Misbehavior) *What kind of emergent misbehavior can result from change notifications and how can it be mitigated?*

To complete the answer of the second part (mitigation) we will now present guidelines for mitigating emergent misbehavior and explain them. In Section 2.5 on page 12 we already answered the first part and the second part partially.

Guideline 1 (Avoid callbacks) *Most of the guidelines are related to callbacks. With normal use of the notification API emergent misbehavior should not occur.*

Callbacks couple an application temporally to configuration changes of other applications or instances of the same application. This observation is the basis for Guidelines 1, 2 and 3. While it is possible with registered variables to check for configuration changes at regular time intervals and react to changes the coupling is not as tight as with callbacks.

Guideline 2 (Wait before reacting to changes) *Waiting after receiving a notification decouples an application from changes and reduces the risk for unwanted **synchronization** (see Section 2.4).*

In applications where applying changes has impact on resource usage (e.g. CPU or disk) applying a time delay as suggested by Guideline 2 is a sensible choice. But this guideline is not only limited to these applications.

Generally waiting before reacting to changes reduces the risk for unwanted synchronization by decoupling the application temporally. Waiting can be implemented using random time delays which further promotes decoupling since applications react at different points in time to changes. Waiting can also be implemented as seen in the case study in Section 4.3 where a flag was used to delay the reaction to changes to the key database configuration.

Guideline 3 (Avoid updates as reaction to change) *Avoid changing the configuration as reaction to a change. This reduces the risk for unwanted **oscillation** (see Section 2.4).*

While Guideline 3 does not forbid updating the key database using `kdbSet()` in a callback it advises to avoid it. If we recall the example from the introduction on page 2 we see how updating as reaction to change leads to unwanted oscillation. If necessary, the function `kdbSet()` should be temporally decoupled as suggested in Guideline 2.

This guideline applies especially to callbacks but is also relevant when variables are polled for changes.

Guideline 4 (Do not use notifications for synchronization) *Applications should not use notifications for synchronization as this can lead to a **phase change** (see Section 2.4).*

Guideline 4 limits the use of the notification API to notifications about configuration changes. There are better suited techiques for other use cases. Applications should not keep track of changes and change their behavior on certain conditions.

For example, this happens when applications synchronize themselves at startup by incrementing a counter in the key database. When a certain limit of application instances is reached the applications proceed with different behavior. If this behavior affects other applications phase change has occured.

Guideline 5 (Apply changes immediately) *Call `kdbSet()` to save updated configuration immediately after users changed configuration. This reduces conflicting changes in the key database.*

When a configuration setting is updated by users within an application Guideline 5 suggests to write the change immediately to the key database using `kdbSet()`. This ensures that other applications have the same view of the key database and operate on current settings.

Guideline 6 (Restrictions within callbacks) *Notification callbacks are called from within Elektra. Calling `kdbClose()`, `elektraNotificationClose()` or `elektraSetIoBinding()` in a callback leads to undefined behavior. The list of restricted functions is non-exhaustive.*

Guideline 6 originates from the case study (Section 4.3) where an incorrectly placed call to `kdbClose()` caused an application crash because the control flow returned from the callback to now unloaded code. While this can be considered an implementation detail it aligns with Guideline 2 since reinitialization of KDB uses more resources than other operations like `kdbGet()` or `kdbSet()`.

4.4 Discussion

We recall our second research question RQ 2:

RQ 2 (Comparison) *How does the use of notifications compare to polling in an application in terms of feasibility, lines of code and CPU usage?*

In order to answer feasibility, practicability, and lines of code for this question we have conducted a case study.

Using only the notification API for target values was feasible in this application. Since use of the API allows the application to implement changes to its configuration at run-time we have also shown practicability.

As we have seen in this case study use of the notification API increased the lines of codes compared to the original compact code for parsing configuration settings. Without convenient functions for converting values as provided by "libautomation" the difference would have been smaller.

The difference in lines of code also originates from the fact that "libautomation" and the new API use different approaches for error handling. Since "libautomation" is built for automation applications which are typically unattended when library functions cannot recover from an error the application is restarted. In contrast, the notification API requires developers to handle errors.

When the semantics of the `elektraNotificationRegister()` functions are known readability is not impacted. Developers have to ensure that updates to variables affect the running application. Using callbacks to respond to configuration changes increases application complexity since callbacks can be called multiple times which requires not only initialization but also de-initialization code.

If and how notifications impact central processing unit (CPU) usage will be answered in the next chapter, "Experimental Evaluation".

CHAPTER 5

Experimental Evaluation

Having shown feasibility and compared lines of code of our notification feature in the previous chapter we now focus on CPU time. In order to complete our answer to research question RQ 2 we will conduct an experimental evaluation and compare two versions of the application from our case study. Then we will examine the measurements and and discuss our findings.

Instructions and code for replicating this evaluation can be found in commit `73ace03` in the "thesis" branch in a fork of the "libautomation" repository[1].

5.1 Test Setup

We will use the "humiditycontrol" application from Chapter 4 and compare two *versions* of the application: one with "polling" updates and one with "notification" updates.

We will measure the CPU time consumed by both versions using the UNIX command `time`. Among other metrics this command measures the number of seconds an application used the CPU in user mode and kernel mode. The *user mode* metric accounts for time the CPU spent for functions of the application and its libraries. The *system time* metric accounts for CPU time spent in kernel mode by the operating system for the application and its libraries. I/O operations like printing a text on the screen typically fall into the second category. Less CPU time spent by application version means the CPU can spend more time on other applications or sleep to save power.

For evaluating the "notification" application version that receives notifications we will use the ZeroMQ transport plugins using the IPC transport. This transport is a local communication transport that does not involve network interfaces like the transmission control protocol (TCP) transport does.

[1] https://repo.or.cz/libautomation/elektra-notification.git, accessed August 2018

The “polling” version of the application updates its configuration using the `kdbGet()` function. It will execute this function at regular time intervals of five seconds. We will call this interval *polling update interval*.

While performing the measurements we will run both versions of the application alternately. An external application will update the numeric “highhumidity” target value configuration setting in the key database with random values. For each *iteration* both versions will be measured with configuration updates in increasing intervals from 1 to 60 seconds. We will call this interval *configuration update interval*.

In order to reduce the impact of application initialization on our measurements the application will receive 20 configuration updates for each iteration. This results in an iteration duration of 20 times the configuration update interval. For example if the configuration update interval is 15 seconds we will measure the CPU time of an application version for a period of 300 seconds (5 minutes).

Everytime an update is applied both versions of the application convert the string value of the configuration setting to a numeric value.

5.2 Results

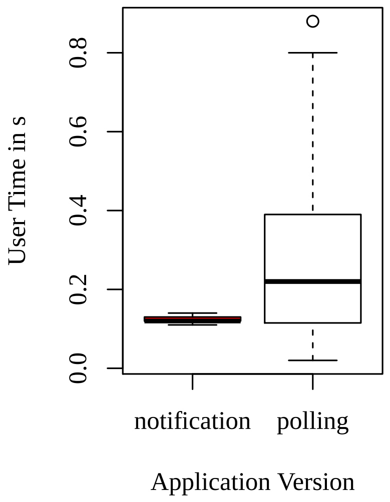

Figure 5.1: Box plot of user times

Figure 5.1 shows box plots for the user time of each application version. The “notification” version has a median of 0.12 seconds and an interquartile range of 0.01 seconds while the “polling” version has a median of 0.22 seconds and an interquartile range of 0.2675 seconds.

Our measurements showed that the system time used by both application versions is negligible with a median of 0 and a interquartile range of 0 seconds. Figure 5.2 shows box

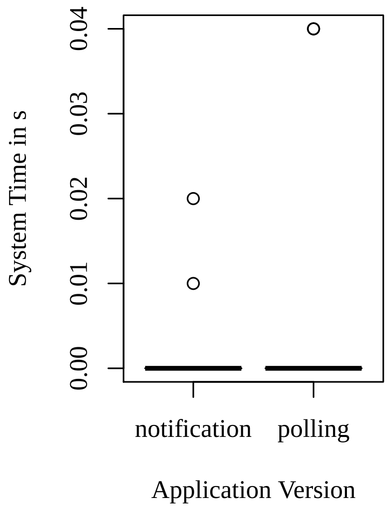

Figure 5.2: Box plot of system times

plots for the system time of each application version. Due to these small numbers the time axis in this box plot has a different scale than the user time box plot in Figure 5.1.

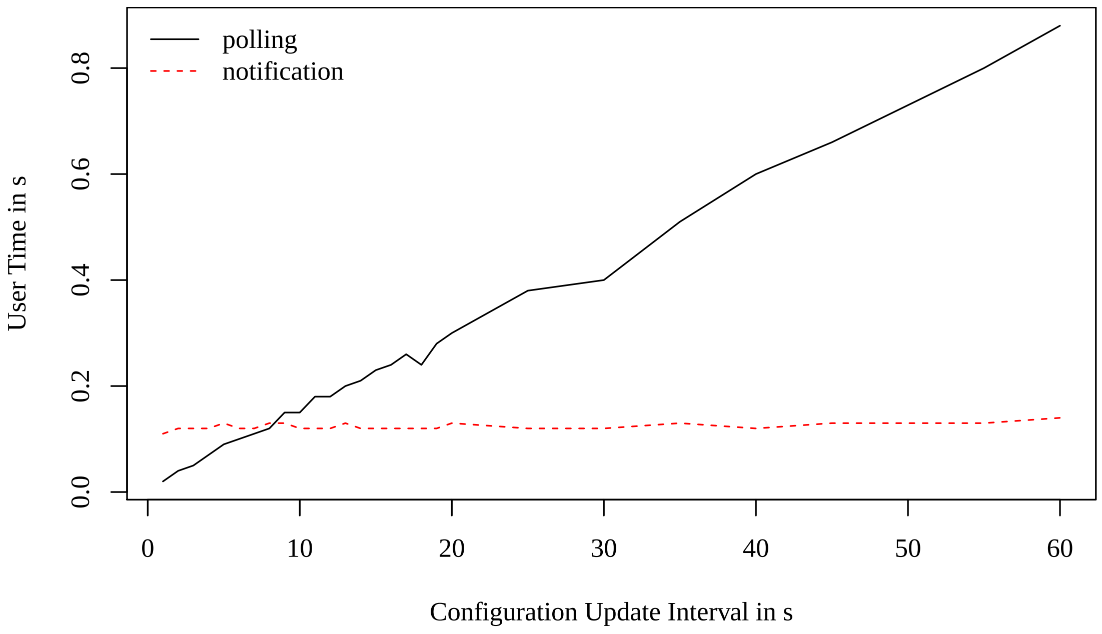

Figure 5.3: Comparison of user times at different configuration update intervals

Figure 5.3 shows lines of both application versions. The solid line shows the user time of the "polling" application version for configuration update intervals from 1 to 60 seconds.

The dashed line shows the user time of the "notification" application version for the same range. The "polling" version shows a linear growth over the range with a minimum of 0.02 seconds to a maximum of 0.88 seconds while the "notification" version remains fairly constant at its median of 0.12 seconds. Between a configuration update interval of 8 to 9 seconds both lines intersect.

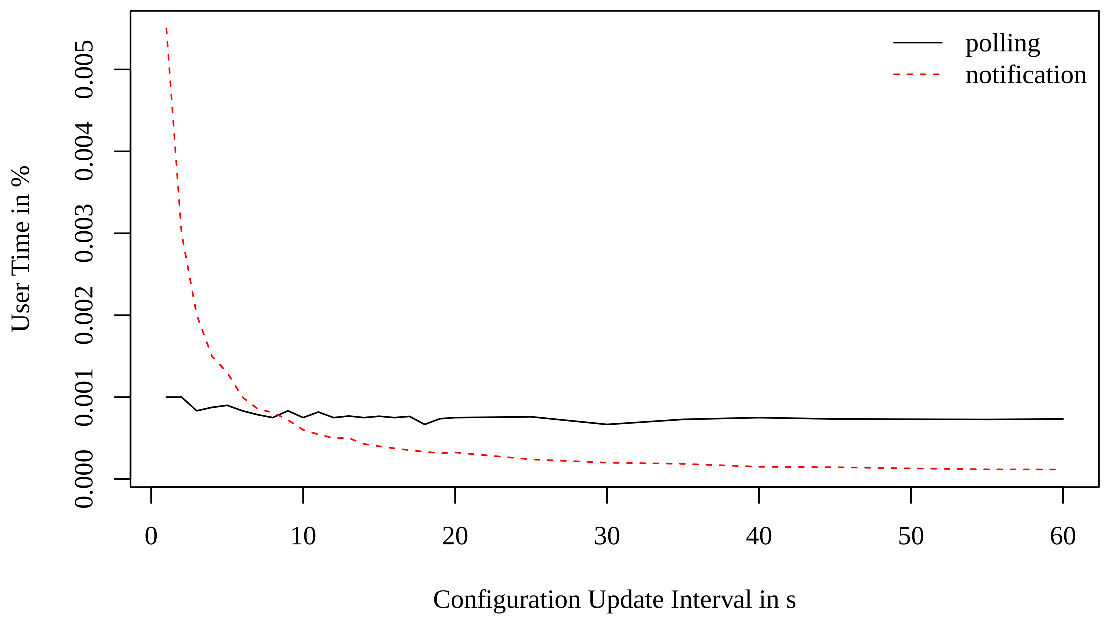

Figure 5.4: Comparison of user times relative to iteration duration at different configuration update intervals

Figure 5.4 is based on the same data as Figure 5.3. In this form of presentation the y-axis shows the amount of user time relative to the iteration duration for both application versions. The "notification" version has an exponential decay with a minimum value of 1.1667×10^{-4} percent. This is 5.7 times less than the "polling" version with the same configuration update interval of 60 seconds. The "polling" version remains fairly constant at a median of 0.00075 percent with an interquartile range of 8.6×10^{-5}.

5.3 Discussion

The lines in Figure 5.3 show that when the *configuration update interval* is less than the *polling update interval* the "notification" version is more efficient in terms of CPU time. This is explained by the "notification" version applying configuration updates less often than the "polling" version checks for updates using `kdbGet()`. The "polling" version checks at a fixed polling update interval regardless of available configuration updates which steadily increases user time as shown in Figure 5.4. Despite Elektra's `kdbGet()`

function checking if the configuration files have been updated before processing them, user time is required for these checks regardless of actual configuration updates.

The presentation in Figure 5.4 shows that the user time of the “polling” version remains fairly constant. Therefore the growth of the “polling” user time is directly proportional to the iteration duration.

The lines in Figure 5.3 also show that when the *configuration update interval* is greater than the *polling update interval* the “polling” version is more efficient in terms of CPU time. This is explained by the “notification” version applying configuration updates more often than the “polling” version. In this case additional updates in the “notification” version are “lost” since the application runs its control routine which uses the updates value less often than updates appear.

The fact that the user times of both versions intersect between 8 and 9 seconds and not when the polling update interval is equal to the configuration update interval at 5 seconds is up to further investigation. This bias likely related to additional initialization costs for the ZeroMQ transport or optimizations within Elektra.

We can conclude that for long running applications with few expected configuration updates notifications are more efficient than polling. When the configuration update interval is greater than a potential polling interval (i.e. an application can only process so many configuration updates per second) polling is more efficient. This can be countered for notifications by applying configuration updates at a lower rate (see Guideline 2 from the case study).

CHAPTER 6

Conclusion and Future Work

In our case study we have shown that by using our notification architecture, applications can register for automated updates to variables on configuration changes. It is also possible to register callbacks that are triggered on configuration changes. This allows applications to implement configuration changes at run-time without restarts. Futhermore our notification architecture supports different transport technologies which can be changed without modification of the involved applications.

After researching current approaches for detection, diagnosis and mitigation of emergent behavior, we have concluded that the required automatic live detection, diagnosis and mitigation of emergent misbehavior can only lower the impact of faulty algorithms, applications or a combination of applications but not repair them. A system that exhibits emergent misbehavior is inherently faulty and remains so even when mitigation is applied. Therefore, we have presented guidelines which both raise awareness for the problems at hand and provide practical advice for application developers.

We have obtained results demonstrating that in scenarios where resource-efficient operation is required and applications are long running, notifications are superior over polling for obtaining configuration updates. Likewise we have found that in scenarios where configuration updates are expected at a high rate, polling is better suited than applying notification updates immediately. In these scenarios the countermeasures suggested by our guidelines can reduce the impact of using notifications.

Under the assumption that configuration is written less often than read, the notification feature should be used instead of polling. The notification feature is obligatory for applications that need to instantaneously implement configuration changes.

We can now revisit the problem related to emergent behavior in the KDE desktop environment[2] introduced in Chapter 1. The workaround implemented by the maintainers was to stop reacting to configuration changes for a certain amount of time. After our research of emergent behavior we can conclude from an outside point of view that waiting was partially the correct solution to stop the occuring oscillation (as per Guideline 2 from Chapter 4). If notifications are distinguishable the solution should ignore only identical notifications and react to new notifications while waiting. The solution would be improved if writing changes was temporarily decoupled as advised by Guideline 3.

The current study was limited by available approaches for automatically diagnosing emergent misbehavior or emergent behavior in general. Further work on this topic would help develop a more involved way of mitigating emergent misbehavior in Elektra.

Future work might focus on evaluating and implementing random time delays upon receiving notifications. Time delays integrated into Elektra will relieve application developers of implementing a correct algorithm that, for example, filters multiple identical notifications.

An open problem for the ZeroMQ transport plugins is that the hub that forwards notifications between applications currently represents a single point of failure. Future work may evaluate whether it is feasible to integrate a hub with automated failover into ZeroMQ transport plugins.

Furthermore we recommend investigation of the use of Redis as storage and notification transport technology for Elektra. Redis provides a feature called *keyspace notifications* which can serve as trigger for implementing a receiving transport plugin when Redis is also used as storage for the key database. Redis also has a publish/subscribe feature which can be used for transport plugins without using it as storage.

After common patterns of applications causing emergent misbehavior in systems of applications using Elektra have been identified, a test suite can be created which simulates other applications. This allows application developers to detect possible problems related to emergent behavior beforehand.

Future work may also include support for the Windows platform, adding an option to use a separate thread for the notification main loop, integrating notifications into a high-level API for Elektra or integrating notifications into Elektra's web-based user interface.

List of Figures

List of Tables

List of Listings

Acronyms

API application programming interface. 16–18, 23, 24, 27, 28, 30–34, 42, 45

CPU central processing unit. 34–36, 38, 39

GUI graphical user interface. 16

IPC inter-process communication. 21, 35

KDB key database. 15, 18, 24–26, 28, 31, 33, 45

SLOC source lines of code. 30

TCP transmission control protocol. 35

Bibliography

[1] Jeffrey C. Mogul. “Emergent (Mis)Behavior vs. Complex Software Systems”. In: *Proceedings of the 1st ACM SIGOPS/EuroSys European Conference on Computer Systems 2006*. EuroSys ’06. Leuven, Belgium: ACM, 2006, pages 293–304. ISBN: 1-59593-322-0. DOI: 10.1145/1217935.1217964.

[2] KDE Community. 2015. URL: https://bugs.kde.org/show_bug.cgi?id=346961 (visited on 08/20/2018).

[3] HashiCorp Inc. 2015. URL: https://github.com/hashicorp/consul/issues/571 (visited on 08/20/2018).

[4] Dave Plonka. 2003. URL: http://pages.cs.wisc.edu/~plonka/netgear-sntp/ (visited on 08/20/2018).

[5] H Van Dyke Parunak and Raymond S VanderBok. “Managing emergent behavior in distributed control systems”. In: *Proceedings of ISA-Tech*. 1997.

[6] Craig W. Reynolds. “Flocks, Herds and Schools: A Distributed Behavioral Model”. In: *Proceedings of the 14th Annual Conference on Computer Graphics and Interactive Techniques*. SIGGRAPH ’87. New York, NY, USA: ACM, 1987, pages 25–34. ISBN: 0-89791-227-6. DOI: 10.1145/37401.37406.

[7] Martin Gardner. “Mathematical games: The fantastic combinations of John Conway’s new solitaire game ’life’”. In: *Scientific American* 223.4 (1970), pages 120–123.

[8] Thomas Moncion, Patrick Amar, and Guillaume Hutzler. “Automatic characterization of emergent phenomena in complex systems”. In: *Journal of Biological Physics and Chemistry* 10 (2010), pages 16–23. URL: https://hal.inria.fr/hal-00644627.

[9] Wai Kin Victor Chan. “Interaction Metric of Emergent Behaviors in Agent-based Simulation”. In: *Proceedings of the Winter Simulation Conference*. WSC ’11. Phoenix, Arizona: Winter Simulation Conference, 2011, pages 357–368. URL: http://dl.acm.org/citation.cfm?id=2431518.2431557.

[10] E. O’Toole, V. Nallur, and S. Clarke. “Towards Decentralised Detection of Emergence in Complex Adaptive Systems”. In: *2014 IEEE Eighth International Conference on Self-Adaptive and Self-Organizing Systems*. Sept. 2014, pages 60–69. DOI: 10.1109/SASO.2014.18.

[11] Pat Dallard et al. “The London millennium footbridge”. In: *Structural Engineer* 79.22 (2001), pages 17–21. URL: https://bcps.org/offices/lis/researchcourse/images/structural_engineering.pdf (visited on 08/20/2018).

[12] J. Fromm. “Types and Forms of Emergence”. In: *eprint arXiv:nlin/0506028* (June 2005). eprint: nlin/0506028.

[13] Claudia Szabo and Yong Meng Teo. “Formalization of Weak Emergence in Multiagent Systems”. In: *ACM Trans. Model. Comput. Simul.* 26.1 (Sept. 2015), 6:1–6:25. ISSN: 1049-3301. DOI: 10.1145/2815502.

[14] F. L. D. Angelis and G. D. M. Serugendo. “A Logic Language for Run Time Assessment of Spatial Properties in Self-Organizing Systems”. In: *2015 IEEE International Conference on Self-Adaptive and Self-Organizing Systems Workshops.* Sept. 2015, pages 86–91. DOI: 10.1109/SASOW.2015.19.

[15] M. Moshirpour et al. “Detecting emergent behavior in distributed systems using an ontology based methodology”. In: *2011 IEEE International Conference on Systems, Man, and Cybernetics.* Oct. 2011, pages 2407–2412. DOI: 10.1109/ICSMC.2011.6084038.

[16] Moez Mnif and Christian Müller-Schloer. “Quantitative Emergence”. In: *Organic Computing — A Paradigm Shift for Complex Systems.* Edited by Christian Müller-Schloer, Hartmut Schmeck, and Theo Ungerer. Basel: Springer Basel, 2011, pages 39–52. ISBN: 978-3-0348-0130-0. DOI: 10.1007/978-3-0348-0130-0_2.

[17] Eamonn O’Toole, Vivek Nallur, and Siobhán Clarke. “Decentralised Detection of Emergence in Complex Adaptive Systems”. In: *ACM Trans. Auton. Adapt. Syst.* 12.1 (Apr. 2017), 4:1–4:31. ISSN: 1556-4665. DOI: 10.1145/3019597.

[18] Chih-Chun Chen, Sylvia B. Nagl, and Christopher D. Clack. “Specifying, Detecting and Analysing Emergent Behaviours in Multi-level Agent-based Simulations”. In: *Proceedings of the 2007 Summer Computer Simulation Conference.* SCSC ’07. San Diego, California: Society for Computer Simulation International, 2007, pages 969–976. ISBN: 1-56555-316-0. URL: http://dl.acm.org/citation.cfm?id=1357910.1358062.

[19] A. J. Oliner and A. Aiken. “Online detection of multi-component interactions in production systems”. In: *2011 IEEE/IFIP 41st International Conference on Dependable Systems Networks (DSN).* June 2011, pages 49–60. DOI: 10.1109/DSN.2011.5958206.

[20] Claudia Szabo and Yong Meng Teo. “An Integrated Approach for the Validation of Emergence in Component-based Simulation Models”. In: *Proceedings of the Winter Simulation Conference.* WSC ’12. Berlin, Germany: Winter Simulation Conference, 2012, 242:1–242:12. URL: http://dl.acm.org/citation.cfm?id=2429759.2430086.

[21] Claudia Szabo and Lachlan Birdsey. “Validating Emergent Behavior in Complex Systems”. In: *Advances in Modeling and Simulation: Seminal Research from 50 Years of Winter Simulation Conferences*. Edited by Andreas Tolk et al. Cham: Springer International Publishing, 2017, pages 45–62. ISBN: 978-3-319-64182-9. DOI: 10.1007/978-3-319-64182-9_4.

[22] M. M. H. Khan et al. “Understanding Vicious Cycles in Server Clusters”. In: *2011 31st International Conference on Distributed Computing Systems*. June 2011, pages 645–654. DOI: 10.1109/ICDCS.2011.73.

[23] Jack B. Reid and Donna H. Rhodes. “Classifying Emergent Behavior to Reveal Design Patterns”. In: *Disciplinary Convergence in Systems Engineering Research*. Edited by Azad M. Madni et al. Cham: Springer International Publishing, 2018, pages 727–740. ISBN: 978-3-319-62217-0. DOI: 10.1007/978-3-319-62217-0_50.

[24] Patrick Th. Eugster et al. “The Many Faces of Publish/Subscribe”. In: *ACM Comput. Surv.* 35.2 (June 2003), pages 114–131. ISSN: 0360-0300. DOI: 10.1145/857076.857078.