PubSub implementation in Haskell with formal verification in Coq

Boro Sitnikovski*, Biljana Stojcevska*, Lidija Goracinova-Ilieva*, Irena Stojmenovska†
*Faculty of Informatics, UTMS Skopje
buritomath@gmail.com, {b.stojcevska, l.goracinova}@utms.edu.mk
†School of Computer Science and IT, UACS Skopje
irena.stojmenovska@uacs.edu.mk

***Abstract*—In the cloud, the technology is used on-demand without the need to install anything on the desktop. Software as a Service is one of the many cloud architectures. The PubSub messaging pattern is a cloud-based Software as a Service solution used in complex systems, especially in the notifications part where there is a need to send a message from one unit to another single unit or multiple units. Haskell is a generic typed programming language which has pioneered several advanced programming language features. Based on the lambda calculus system, it belongs to the family of functional programming languages. Coq, also based on a stricter version of lambda calculus, is a programming language that has a more advanced type system than Haskell and is mainly used for theorem proving i.e. proving software correctness. This paper aims to show how PubSub can be used in conjunction with cloud computing (Software as a Service), as well as to present an example implementation in Haskell and proof of correctness in Coq.**

***Index Terms*—cloud computing, Software as a Service, PubSub, Haskell, Coq**

I. Introduction

A cloud can be both software and infrastructure. It can be an application accessed via the Internet or a server provided when needed. If a service can be accessed by a device, regardless of the operating system of that device, then that service is cloud-based [1].

Typically, three criteria are defined as labels whether a particular service is a cloud service [1]:

- the service is available through a web browser or web services API,
- zero capital spending is needed to get started,
- payment is required only for those services that are used.

The PubSub (publish-subscribe) pattern allows for easy message transfer to specific channels.

Haskell and Coq are programming languages designed with an aim to accomplish software correctness, based on a typed version of the lambda calculus system [2], [3], [4].

II. Architecture

A. Cloud computing

We can categorize most cloud-computing projects in three basic categories:

- projects that deploy services for multiple applications or clients,
- projects that are single, standalone cloud applications,
- cloud-based service provider projects (e.g. Google).

Besides these project categories, there are three main cloud-based architectures [5]:

- SaaS - software as a service (e.g. Google Apps, Salesforce, Dropbox, games such as World of Warcraft)
- PaaS - platform as a service (e.g. Windows Azure, Heroku, Google App Engine, WordPress web site development platform)
- IaaS - Infrastructure as a Service (e.g. Google Cloud Platform, Amazon Web Services, Microsoft Azure, etc.)

B. Software as a Service

SaaS refers to software hosted on the cloud in a central location (Figure 1) [1]. Typically, this architecture consists of web-based software, but it is not limited to.

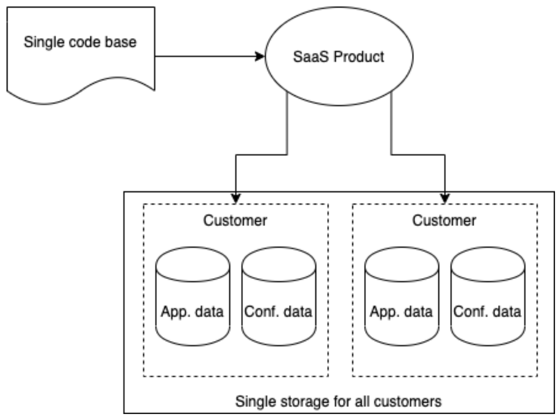

Fig. 1. SaaS architecture

SaaS applications are accessed through a client such as a web browser [1]. This architecture applies to many business applications, including Enterprise Resource Planning (ERP), Customer Relationship Management (CRM), Office software, messaging software, etc [5]. SaaS is involved in the strategy of almost all leading software companies - Amazon, Google, Microsoft [5].

C. PubSub service

The PubSub pattern is a messaging pattern where publishers can send messages to specific channels to which subscribers are subscribed [6]. The commands SUBSCRIBE, UNSUBSCRIBE, and PUBLISH implement this pattern. This separation of publishers and subscribers provides better scalability to the service [6].

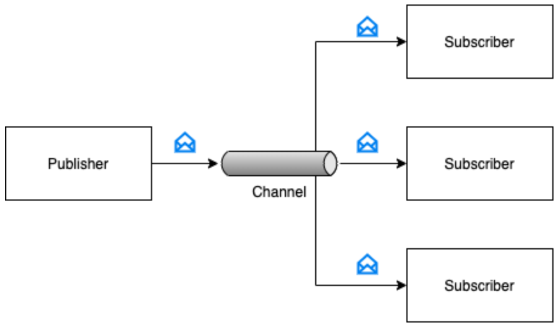

Fig. 2. PubSub architecture

There are three basic components to understanding a PubSub messaging scheme, as shown in Fig. 2:

- Publisher - Publishes messages to the communication infrastructure
- Subscriber - Subscribes to a specific channel/category of messages
- Information Infrastructure (Channel) - Handles subscriptions and receives publisher's messages

III. Implementation in Haskell

In our implementation of PubSub, we use the Haskell programming language. Haskell is an advanced functional programming language. The development of Haskell is rooted in mathematics and computer science research [2].

A. *Subscription module*

This module represents the business logic functions of the PubSub architecture. Namely, each subscription is represented as a pair of Channel and Connection. Further, all such subscriptions are merged into a single list of subscriptions.

The function getHandlesByCh takes a Channel and a list of Subscriptions and then returns a filtered list of subscriptions such that the channel is matched. The Haskell's built-in function filter [2] will be used by the publish command.

```
getHandlesByCh :: Channel -> [Subscription a]
    -> [Subscription a]
getHandlesByCh c = filter (\x -> c == fst x)
```

The function removeHandleByCon takes a connection and a list of Subscriptions and then returns a filtered list of subscriptions such that the selected connection is not contained. This will be used by the quit command.

```
removeHandleByCon :: (Eq a) => Connection a
    -> [Subscription a] -> [Subscription a]
removeHandleByCon h = filter (\x -> h /= snd
    x)
```

The function removeSubscription takes a channel and a connection and a list of Subscriptions and then returns a filtered list of subscriptions such that the selected connection and channel are not contained. This will be used by the unsubscribe command.

```
removeSubscription :: (Eq a) => Channel ->
    Connection a -> [Subscription a] ->
    [Subscription a]
removeSubscription c h = filter (\(x, y) ->
    not (x == c && y == h))
```

The function addSubscription takes a channel and a connection and a list of Subscriptions and then returns a list such that the selected connection and channel are contained. This will be used by the subscribe command.

```
addSubscription :: Channel -> Connection a ->
    [Subscription a] -> [Subscription a]
addSubscription c h s = (c, h) : s
```

The function hInSubscription checks if a given channel/subscription is contained into a list of subscriptions.

```
hInSubscription :: (Eq a) => Channel ->
    Connection a -> [Subscription a] -> Bool
hInSubscription c h = any (\(x, y) -> x == c
    && y == h)
```

The next function is a helper function for the subscribe command.

```
subscribe ch h s = if not (hInSubscription ch
    h s) then addSubscription ch h s else s
```

Conversely, the following function is for the unsubscribe command.

```
unsubscribe ch h s = if hInSubscription ch h
    s then removeSubscription ch h s else s
```

Finally, the last function publish uses the condition the types t and m to be Foldable (to be iterated) and Monad (to be chain-operated, for example in IO) respectively. This covers a more general case.

However, in this specific case (PubSub), mapM iterates through the list of subscriptions and perform the function f on each subscription individually. In the PubSub implementation, f is a function that writes to an IO object (hPutStrLn). This generalization, even-though complex, is necessary to easily prove correctness, since in Coq, IO does not exist as a concept.

```
publish :: (Traversable t, Monad m) => (t2 ->
    m b) -> t (a, t2) -> m (t b)
publish f = Data.Traversable.mapM (\(_, y) ->
    f y)
```

B. *Main module*

This module represents the entry-point of the program, where the logic to accept new clients is initialized.

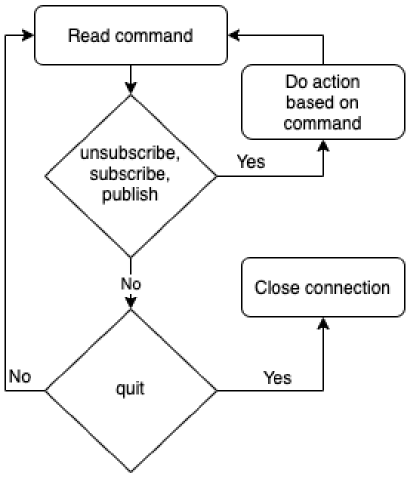

Fig. 3. PubSub algorithm

The main module accepts new connections indefinitely, launching a new thread for every connection. These threads use the algorithm described in Fig. 3.

The algorithm handles different commands passed through the communication channel. For a given command, list of subscriptions, and a connection, the algorithm updates the list of subscriptions and returns a boolean result, representing the success of the command's execution. Accepted commands are `unsubscribe`, `subscribe`, `publish` and `quit`. This algorithm will be executed recursively until the client closes the connection.

C. Running example

In this subsection, we will demonstrate the usage of our application through the client program `telnet`. We will create two connections, subscribed to channel 1 and 2 respectively, and then we will send messages to these subscribers through a third connection.

First connection:

```
$ telnet localhost 123 import
Connected to localhost.
Write 'publish <ch> <msg>' to publish,
    'subscribe <ch>' to subscribe.
> subscribe 1
```

Second connection:

```
$ telnet localhost 123
Connected to localhost.
Write 'publish <ch> <msg>' to publish,
    'subscribe <ch>' to subscribe.
> subscribe 2
```

Third connection:

```
$ telnet localhost 123
Connected to localhost.
Escape character is '^]'.
Write 'publish <ch> <msg>' to publish,
    'subscribe <ch>' to subscribe.
> publish 1 hey there
> publish 2 hello!
```

The first and the second connection will receive `[1] hey there` and `[2] hello!` respectively.

IV. Formal Proof of Correctness in Coq

Coq is a programming language used as an interactive theorem prover that enables expressing mathematical definitions [3]. By mechanically validating evidence of mathematical claims, Coq helps in producing a certified program. It is based on the theory of Calculus of Inductive Constructions, a system in the lambda calculus family [4]. Haskell is also based on a weaker version of lambda calculus, hence there is a strong connection between these two programming languages.

Coq's initial release was done in 1989 by INRIA [3]. This programming language has support for representing dependent types [7]. These types are precisely what enables us to write mathematical proofs [8].

Haskell has no support for dependent types, so the first challenge that we face is how to prove some basic property of our programming code (written in Haskell) in Coq. For this, we use an existing tool called hs-to-coq [9] that allows us to convert Haskell code to Coq. The reverse conversion is already supported in Coq itself. Using this tool, we get the `Subscription.v` (Coq source code) file.

Semantically, this auto-generated file from hs-to-coq contains the same functions that we have defined earlier in Haskell. The only difference is the syntax, where the Coq syntax is used instead of Haskell. At this point, we can start using the power of Coq.

Thus, we create the following `Proofs.v` file that has the following content:

```
Require Import Prelude.
Require Import Subscription.
Require Import Proofs.GHC.Base.
Require Import Data.Semigroup.
```

Now we can finally work on the proof itself. We will prove a few simple properties for this paper's argument. We first define a list of subscriptions to use in our proofs:

```
Definition subs := addSubscription #1 &"fh01"
   nil.
```

We then proceed showing that
$\mathtt{length}(\mathtt{getHandlesByCh}\ 1\ \mathtt{subs}) = 1$

```
Lemma example_1 : GHC.List.length
    (getHandlesByCh #1 subs) = 1%Z.
Proof.
  auto.
Qed.
```

We will explain the syntax used in the proof briefly. The interested reader can look at the details in [10].

In the code above we have proved the lemma named `example_1` which states that the length of the evaluation of `getHandlesByCh 1 subs` is 1. We begin the proof using the `Proof.` command. What follows afterward are commands called tactics. These are macro commands that use the lambda calculus rules to simplify formulas. In this case, by using the `auto` tactic, Coq can mechanically prove the claim, because the claim is simple enough. If the claim was a bit more complex, we would have to use different tactics.

We run Coq and it executes the code mechanically, returning no errors meaning the proof is complete.

```
Lemma example_2 : GHC.List.length
    (getHandlesByCh #2 subs) = 0%Z.
Proof.
  auto.
Qed.
```

The lemma `example_2` is similar to `example_1`. It claims that the subscription of channel 2 does not exist in the list, namely that the length of such a list is 0.

```
Lemma example_3 : getOption (publish (fun y ⇒
    GHC.Base.return_ 1) subs) = Some (1 ::
    nil).
Proof.
  auto.
Qed.
```

The lemmas `example_3` and `example_4` prove that the second argument of `publish` has an effect on the output of the subscriptions. This is as expected, since this is how we defined `publish` in Haskell earlier.

```
Lemma example_4 : getOption (publish (fun y ⇒
    GHC.Base.return_ y) subs) = Some (&"fh01"
    :: nil).
Proof.
  auto.
Qed.
```

With the following lemma we will prove the fact:

$\forall l, \mathtt{hInSub}(1, \mathtt{"fh01"}, (\mathtt{addSub}, 1, \mathtt{"fh01"}, l)$

That is, for all lists `l`, upon which a subscription 1 with `"fh01"` is added, the function `hInSubscription` returns true.

We further take a brief look at the kinds of errors that Coq may return.

```
Lemma example_5 : forall l : list
    (Subscription Base.String),
    hInSubscription #1 &"fh01"
    (addSubscription #1 &"fh01" l) = true.
Proof.
Qed.
```

The example given above results with the following error from Coq:

```
Error: (in proof example_5): Attempt to save
    an incomplete proof
```

It warns us that the proof is not complete. We can use the command `Show Existentials` to see the current state of the proof:

```
Existential 1 =
?Goal : [
        |- forall l : list (Subscription
             String),
           hInSubscription 1%Z &"fh01"
             (addSubscription 1%Z
               &"fh01" l) =
           true]
```

This proof is also simple enough for Coq, so we can use `auto` to complete it as well.

CONCLUSIONS

Moving to the cloud is one of the current challenges in enterprises. This technology provides a new ”on-demand” paradigm for information and communication technologies.

The advantages are cheaper systems, access from countless devices, centralization of data. The main disadvantage is that an Internet connection is required.

PubSub is just one of the many SaaS possibilities. As the usage of such systems continues to rise, formally proving correctness according to specifications is crucial to the system functioning and accuracy as expected. The solution presented in this paper demonstrates how Haskell in conjunction with Coq can be applied to perform this vital step in the process of cloud-based software development.

REFERENCES

[1] G. Reese, *Cloud Application Architectures*, O'Reilly, 2009.
[2] B. O'Sullivan, D. Stewart, and J. Goerzen, *Real World Haskell*, O'Reilly, 2008.
[3] INRIA, ”The Coq Proof Assistant”. [Online]. Available: https://coq.inria.fr/ (Accessed Jan. 2020)
[4] A. Church, “An Unsolvable Problem of Elementary Number Theory,” *American Journal of Math.*, 1936.
[5] G. Shroff, *Enterprise Cloud Computing*, Cambridge University Press, 2010.
[6] Redis, ”Pub/Sub”. [Online]. Available: https://redis.io/topics/pubsub (Accessed Dec. 2019)
[7] A. Chlipala, “An Introduction to Programming and Proving with Dependent Types in Coq,” *Journal of Formalized Reasoning Vol. 3, No. 2*, 2010.
[8] B. Sitnikovski, *Gentle Introduction to Dependent Types with Idris*, Leanpub/Amazon KDP, 2018.
[9] A. Spector-Zabusky, J. Breitner, C. Rizkallah, and S. Weirich, “Total Haskell is Reasonable Coq,” *Proceedings of the 7th ACM SIGPLAN International Conference on Certified Programs and Proofs*, 2018.
[10] INRIA, ”The Coq Reference Manual”. [Online]. Available: https://coq.inria.fr/ (Accessed Jan. 2020)