Distributed systems in Haskell, the P2Pmedia application

Han-Hui Chiao, University of Melbourne, hchiao@student.unimelb.edu.au

***Abstract*—This report asserts that the programming language Haskell is ready for real world distributed applications. It discusses challenges which distributed systems face, and shows that actor model and pure code are the keys in building a reliable and robust distributed application. By using Cloud Haskell and Yesod, we present P2PMedia, a peer to peer social media platform that is designed to build tighter bonds between individuals in a community. By developing the application, we demonstrate that the overall Haskell environment is mature. With the expressive and pure nature of the language coupled by the strict constraints of actor model, Haskell produces simple yet robust, elegant yet powerful real world distributed system.**

***Index Terms*—Peer to peer, Cloud Haskell, Yesod, Distributed systems**

I. Introduction

THROUGHOUT the ages, computational speed has increased while prices have decreased resulting in an abundance of computational power as a whole [5]. Therefore, utilizing all this computational power efficiently is paramount.

The prime goal for distributed systems is to utilize many computers to achieve services that a single computer cannot accomplish or might execute poorly. The idea is that individually, one's ability is limited. However, when working as a group the achievements could be unlimited. It is universally accepted that the key challenge for effective team work is communication and coordination. The communication area has been widely researched and comprehensively refined due to the explosion of internet in the past two decade. The technology is relatively mature and the capabilities and boundaries are well known. However, coordination presents challenges, and managing the complexity of coordination is nontrivial. This report will be focused on the latter.

We have developed the P2PMedia application to demonstrate how, by using actor model and purity, we are able to manage the complexity of coordination well, and achieve a better quality distributed system. The high level architecture of the application is to create interconnecting cloud of peers that provides knowledge for individuals of that cloud. The idea is to store "local" knowledge that is most beneficial for the cloud, thus the cloud would organically and incrementally improve as each peer contribute different information from experiences. Moreover, unique niche knowledge would be explored and preserved, creating diversity and a sense of community among participants.

II. Distributed System Challenges

One of the main goals for a distributed system is, from a user's prospective, that the system should be viewed as a single unified unit. The current popular term would be cloud; one does not need to be concerned with how many computers are running or where the computers exist, or even how they are connected. All that the user has to know is how to use the service provided by the system to complete the individual's desired tasks; the nuts and bolts would not be exposed to the user. To achieve this, the system has to fulfill criteria such as resources accessibility, distribution transparency, openness, scalability, security, performance issues, and quality of service [11], [16]. However, fulfilling these conditions has proven to be somewhat challenging as unpredictable bugs occur due to the distributed nature of the system.

> "A distributed system is one in which the failure of a computer you didn't even know existed can render your own computer unusable" –Leslie Lamport

A. The Unified Objects approach[22]

The unified objects approach is a common design and architecture for distributed system. The principal idea is, as a programmer, objects in a remote machine should act as any other local objects, and the mechanism for achieving this should be abstracted out from the programmer. In other words, there should be essentially no distinction between objects that share an address space and objects that are on two machines with different architectures located on different locations worldwide. Whether a call is local or remote should have no impact on the correctness of a program. Thus, programmers can avoid the details of interfacing with the network, and focus solely on the work at hand.

Many frameworks have been designed with this methodology in mind. Famous ones are ONC RPC, Java Remote Method Invocation (Java RMI) and Common Object Request Broker Architecture (CORBA); they are used in industry to various degrees.

1) Why Unified Objects approach does not work[22]: In a Unified Objects approach, in order to abstract out the difference of a local object and a remote object, masking the details of network and memory transport is necessary. However, what really makes distributed systems challenging compared to local systems is dealing with partial failure of components and the lack of a central resource manager [22]. Partial failure is a fundamental hurdle that needs to be resolved in a distribution system e.g. lost connection. To elaborate, both local and distributed systems have components that will fail some of the time e.g. database query. In a local system if a partial failure occurs, the failure will either affect all other components or be detected by some central resource allocator,

for example exception caught by Java Runtime Environment, whereas in a distributed system, when one component fails e.g. a node crashes, the others will still continue executing. There is no exterior component that can detect failure and coordinate other nodes to take actions in respect to the failure. Moreover, there is no way of knowing the nature of the failure, which could be either node crash or network failure [22]. In a service that increments a global counter, if one of the worker node fails, the master node would not know if the node had crashed or the message was lost in the network. The logical way to recover from this is to resend the request; however, retrying could produce a result ranging from 0 to 2, when only 1 would be correct[20]. Unified objects approach does not allow the programmer to tackle this complication, as it violate its core principles, thus producing unreliable systems that will behave erratically and unpredictably when deployed in production.

B. Explicitly dealing with network and memory transport

Manually handling network and other partial failures is needed to achieve better quality distributed systems. Thus, the actor model has been adopted by many frameworks and languages to build better distributed systems.

1) Actor model[14], [7]: A different way of handling interaction between components is the actor model. In this methodology, every component is an actor. Each actor has business logic and a mailbox. Communication between actors is done by sending messages. Actors will react accordingly depending on what messages they receive. Mailbox will buffer messages if they cannot be processed immediately. When receiving and processing a message, the actor can do a variety of actions such as send messages to other actors; it can spawn new actors, or even alternate their behaviors when receiving the same message again.

This model introduces two important designs properties, the asynchronies of communication between components, and the lack of shared state between components[21]. Firstly, the asynchrony implies that once the sender sends its message it will continue execution without waiting for a reply. This is critical due to the fact that the system will function whether or not the message is received therefore fault tolerance is built into every component in the system. Secondly, the lack of shared state between each actor allows for failures of actors in a way that potently would not alter or interfere with other actors. Message passing is the only way an actor can communicate with another actor, giving individual actors full control of their internal state. The design choice of only message passing removes the need for locks thus avoiding complications like lost-update problem and other undesirable problems associated with locks. Essentially every actor is a miniature process. The property of safely allowing actors to fail greatly increases the robustness of distributed systems and decreases potential bugs.

2) Erlang [21], [9]: Erlang is a language that was designed explicitly for concurrency, distribution and scalability. In Erlang the actor model plays a prominent role in the overall design of the language. Every component is an actor. Even if building a local application, the components would still communicate via messaging.

Erlang has been in the industry for over 20 years and have proven to be exceptional at building distribution systems. Many large and well known application is developed in Erlang, the following are a some popular ones, Facebook Chat system [19], GitHub[10], WhatsApp[1], and Amazons SimpleDB for EC2 [9].

3) Debugging hell: Managing the details of all the interactions is very challenging, and covering all cases in a system with millions of nodes is next to impossible. Most of the time the system would be working correctly, however, there are times when happen by chance that the interacting components align absurdly as to cause the system to be stuck in an undesirable state. One would think that since it happens by chance therefore the bug is an isolated case thus can be patched up. However, the problem is that there may be multiple "isolated" cases that happen just often enough to render the whole system highly unstable and unreliable. This is a nightmare scenario for any programmer. For one isolated case, even before debugging, recreating the error would be incredibly tedious. The programmer would have to reenact the exact environment in which the error occurred, and in a distributed system that is not a trivial task. Imagine a distributed system with millions of nodes; let's assume that if twenty nodes executing a method at the same time would cause the system to be stuck in a deadlock. How is the programmer going to find this bug? Maybe he could isolate only a few hundred nodes out and use brute force to test out all the permutations? This approach would take a long time, and with no guarantee that the nodes that were isolated are the nodes that would cause the error. Even if the programmer is able to find the error and fix that specific bug, potentially there may be hundreds more bugs that could happen due to the vast amount of nodes and interaction between nodes.

The core of the problem lies in indeterminism[20]. Indeterminism is defined as "a theory that the will is free and that deliberate choice and actions are not determined by or predictable from antecedent causes"[17]. In programming terms, it means that a function not only returns a value, it also does unpredictable side effects[15]. The problem with this is it is hard to reason with what the function is actually going to do, the function could simply just return a value and do logging, or it could return a value and fire a missile! The caller of the function would not be able to anticipate what all the outcome would be, therefore the caller can only react with a variable degree of certainty, never fully sure of all the consequences. In a local system, it is easier to handle side effect, since all the components involved are known, so it is possible to consider all the possibilities and act accordingly, whereas in a distributed environment, multiple nodes are connecting and disconnecting at the same time, and it is exponentially harder to do the same, as no node would be fully aware of all other nodes. Thus, indeterminism is a major concern when designing a distributed system.

As demonstrated previously, side effects are not desirable in a distributed system. To further the argument, consider a function that reads a global state and returns the value. If some

other function or functions modify the state then the outcome of the original function would change. For the original function, even when using the same input parameters, the return values are not the same. This function has side effect and is erratic and unpredictable. Side effects effectively add invisible inputs and outputs to a function, resulting in a prime source of bug. Furthermore, many bugs are caused by unanticipated side effects; even more are due to misunderstanding circumstances in which functions may return different results for the same inputs[18]. Thus, side effect is a prime adversary for robust and reliable application.

C. Purity[13], [18]

Purity is a simple but powerful philosophy of coding. It is putting constraint on what you can and cannot do in order to achieve robust, reliable and modular code. We call a function pure when, with the same input arguments, the output result will be identical; ultimately a function without side effects[18]. This property is called referential transparency. It gives us guarantees and assurances of what exactly the pure function can and will do[18]. A good analogy would be the following: in a modern lawful society, we have the guarantee that if we spend our time building something, a business for example, it will not be taken away from us due to some robbery or warlord threatening our life. With this assurance, people can spend most of their effort in building the economy and society rather than worrying about fundamental problems such as security and rights. Essentially obedience to laws and constraints will result in better outcomes for society as a whole.

When a function is pure, the programmer can clearly understand and anticipate how and in what way a function is going to behave. When it's pure, the code would not depend on the value of a global variable, or the contents of a database, or the state of a network connection[18]. The programmer does not need to take into the account what the state of a global variable is or even some other function's state, the programmer only needs to consider the input arguments that the function takes. Thus, the mental power needed to reason with the function is dramatically reduced, allowing the programmer to focus solely on the problem that the function is trying to solve. By obeying to purity, the programmer can freely develop components, fully confident that the component would not affect other parts of the system and vice versa.

Purity simplifies complexity, and in a distributed environment the less complexity the better, the more assurance the better. If most parts of the distributed system were coded in a pure fashion then when and if a bug occurs, it would be substantially easier for the programmer to reason and find the bug, as only a small part of the code could result in the error.

For most languages like Java, C or even Erlang, writing pure code is entirely up to the discipline and experience of the programmer. There is no guarantee from the compiler that a function would return the same results if the input arguments were to be the same. Likewise, there is no guarantee that a function has no side effect. The only way to know what side effects a function will have is to read its documentation and hope that it is accurate and nothing has changed since it was written. Wouldn't it be great if we could be certain of pure code? Enter Haskell.

1) Haskell[18]: Haskell is a general-purpose purely functional programming language. It has many different features compared with conventional languages, features such as: purity, strong static type system, lazy evaluation, pattern matching, list comprehension, type classes, monads and type polymorphism.

In Haskell, all codes are pure with the exception of monads, the pureness is guaranteed at compile time. This purity constraint is achieved via the strong type system, therefore, once the programmer declares, in the signature that a function is pure, there is no way the programmer can accidentally put impure side effect code into it. The code will refuse to compile if otherwise.

Nevertheless, inevitably, all programs needs to interact with the outside world in order to be useful, thus, we have to introduce side effect at some point. By using monads, Haskell cleverly allows side effect in a constrained manner. As a consequence of this design, we now have a clear divide of functions with side effects and those without. This encourages the programmer to write the majority of the code in pure functions and only introduce side effect code should it be absolutely essential. In this style of coding, the parts of the code that execute side effects are simple, leaving all the complexity in the pure side of the code. When code is written with this discipline, impure code is isolated and simple, resulting in a small margin for error caused by side effects[18].

Due to the purity that is baked into the language, Haskell is a prime candidate with which to build distributed systems.

2) Choosing Cloud Haskell[12]: After analyzing the difficulties of programming in a distributed environment, one can conclude that purity and actor model should play a major role in achieving a reliable system, thus, choosing Cloud Haskell[12] becomes a logical choice. Cloud Haskell is a Haskell library that is built to implement the actor model. The authors closely followed the design of Erlang but added purity, types, and monads to utilize the advantage of Haskell. The default purity of Haskell is a powerful ideology that keeps the programmer in check; whereas the overall actor model dictates the interface in which components interact, further reducing the corruption between components. Therefore, Cloud Haskell is idea for creating distributed systems.

III. Haskell Environment

One of the main challenges of using exotic languages such as Haskell is the lack of a mature developing environment. However, we have found it extremely pleasant as detailed below when developing our application. Haskell has a vibrant online community, thousands of libraries centralized on HackageDB[2] and easily searchable on Hoogle[4] and Hayoo[3], the package manager Cabal is also pleasant to use. We will discuss the two major library and framework that's been used for the P2PMedia application in the following section.

A. Cloud Haskell

1) Node, Process & Message passing: In Cloud Haskell, an actor is called a process, and communication between processes is performed by message passing. Cloud Haskell implements a process by spawning a light-weight thread for each process just like in Erlang. However, Cloud Haskell also allows threads to be spawned inside a process which Erlang cannot. This gives the programmer more freedom to perform computations and potentially speed up some algorithms by computing parts of the procedure in parallel. Due to monads, the threads inside a process will not be able to interact with anything outside of the process insuring that side effects stay constrained. Nodes are individual address spaces where processes reside. Because processes communicate via message passing, communication between two processes that are in the same node or in different ones do not have any differences, thus neatly conforming to the actor model.

2) Strong Type Messaging[12]: In the spirit of Haskell, Cloud Haskell introduces strong type message passing that differs from Erlang. Strong type message passing offers the assurance and guarantee of the exact type of message that a specific channel can send. For example, a process asks another process for a user algebraic data type[18]. The first process will send a normal message containing the channel to the latter process, and then the latter process will send the user information back via the channel. If by accident some other information (e.g. password algebraic data type) was sent back, the error would be caught at compile time. Therefore, strong type messaging insures increased robustness and makes certain types of bugs virtually impossible to occur.

B. Yesod

Yesod is a popular web framework in Haskell; it follows the Representational State Transfer (REST) software architecture, and runs on an extremely fast warp web server[23]. In fact Yesod running on warp is exponentially faster than other popular interpreted languages web framework such as Ruby on Rails and Python's Django [23]. The strict type system of Haskell also becomes an ally in creating a better web interface, generating proper links, avoiding XSS attacks, and dealing with character encoding issues, all made easier in a strictly typed environment. The following will introduce some important aspects of the framework.

1) Handlers: Due to the RESTful design of Yesod, every resource is handled by a separate handler. In a typical web application, a web page is the most common resource that has been used; therefore each web page will have its own handler serving http request / response to the web browser. The handler would host all the business logic for the web page. Other actions such as database querying would also be allocated here.

2) URL Routing, Html, CSS and JavaScript: For URL routing, Yesod has an intermediate data type that utilizes Haskell Strong Type to safeguard from broken links, thus broken links are also detectable at compile time. Hamlet (HTML), Cassius (CSS), Lucius (CSS), Julius (JavaScript) are mini template languages that represent the standard web languages respectively, as a whole they are the Shakespeare template family. There are benefits to using these languages. They add minimum interfaces to the underlining language while providing greater flexibility and power, giving compile time guarantees, using strong type that deters XSS (cross-site scripting) attacks, automated checking of valid URLs, and interpolation for creating dynamic content.

3) Database: Yesod supports a large variety of database back-ends. Database such as PostgreSQL, SQLite and MongoDB, CouchDB and MySQL are fully supported. Furthermore, Yesod places an intermediate layer in between the application and storage layer. It uses a special syntax for query commands that will be translated to the native command of the back-end, be it SQL or otherwise. This decoupling of application and database allows the programmer to focus on the problem in hand and not worry about the complex details of different back-ends.

IV. Real World Problem and Solution

We have developed a web interface distributed application to demonstrate development in a Haskell environment, show casing actor model, purity and modern web libraries working in tandem, and in the process demonstrating the maturity of the Haskell environment, thus supporting the idea that Haskell is ready for real world applications.

A. The problem to solve

In the past two decades the explosion of internet has greatly shorten the distance between people and literally made the "global village" vision a reality. However, we have observed that while the leaps and bounds of technology have connected people far and wide, it has neglected or in some cases pushed apart traditional communities, communities such as neighborhood, classes, and colleagues. We argue that whilst having connections with people from all over the world is paramount in this day and age, having close connection with people that are immediately close to you is just as, if not more, important.

B. Solving the problem with P2PMedia

To solve the problem, we identify some key functionality that the service should provide. First it must focus on connecting people that are close to each other. Secondly it should provide a centralized area for socializing between community members. Thirdly, it will be considerably more effective if a sense of responsibility and reliance on each individual in the community is generated in order to promote a higher level of participation.

With the previous design consideration in mind, we present the P2PMedia application. P2PMedia is a peer to peer based social media application that currently provides two features. First is the Twitter-like messaging service, and second is a Wikipedia-like service. The major difference, compared to traditional social media, is the decentralized nature of the application. Since the application is decentralized, the system as a whole will run more robustly, even in scenarios such as

90 % node failure, the system would still function correctly. Another advantage of this model is the lack of reliance of some major corporation that you have to trust in keeping the system running properly and correctly. Moreover, personal information and data is secure in the individual's machine potentially resulting in less security concerns or possible exploitation.

V. P2PMedia Architecture

There are two main components in the application: the handlers and the p2p module. The p2p module is further split up into Controller and Listener. The two reside on the same node thus they have the same network address. However, they run in different processes thus they do not have shared memory and have to communicate via message passing. The Controller deals with business logic regarding peers, it receives commands from Handler (via messaging) and takes actions accordingly. Some actions may be sending messages to other peers asking for information; others may be sending messages back to handler giving error messages. The Listener is like the peer's server, it has access to the standard output of the peer, and it is also capable of executing database commands.

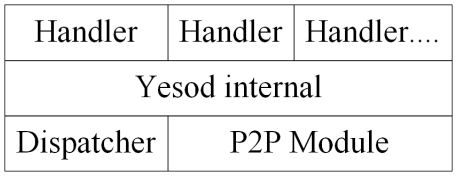

Figure 1. P2PMedia Architecture

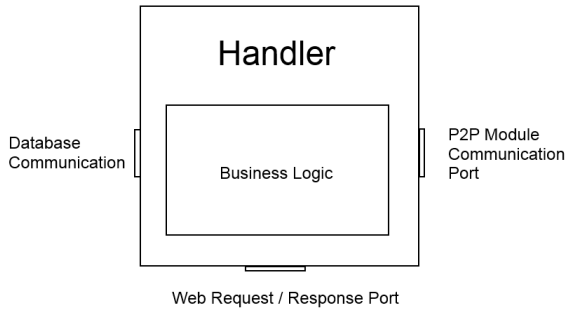

Figure 2. Handler Architecture

A. Major design consideration

In Yesod, when web browser sends an http request via a specific rout, the request will be routed to the handler function that has been assigned to this specific rout. The way this is accomplished is done by the Yesod dispatcher, and the details

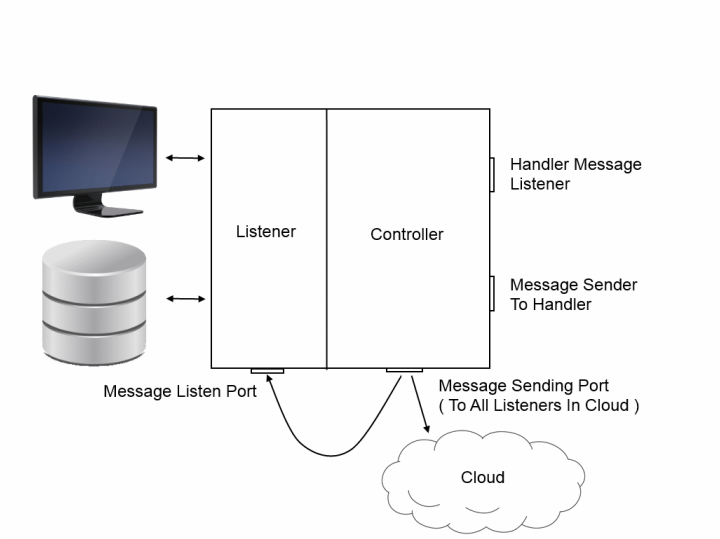

Figure 3. P2P Module

of the inner working are not in the scope of this paper. Once the request is sent to the handler, the handler will execute its business logic and return HTML, CSS and JavaScript files for the browser to display.

Due to the architecture of Yesod and the Haskell purity concept, a major problem occurs when the handler needs to interact with the p2p module. The only input for each handler function is the http request, and due to the purity concept in Haskell, the programmer cannot use side effects to manipulate code inside the function handler. However, Haskell allows certain side effects in monads, but only side effects that are specifically allowed in that specific monad. In the Yesod monad Handler, it allows IO actions and database actions.

After identifying the problem and understanding the tools that are available to solve the problem, there seems to be three approaches in solving the issue.

The first idea would be to find a way inside the Yesod framework to propagate the information from P2P module to the handler in question via http request. However, it is undesirable to alter the internals of a fully functional framework where the alterations may have ramifications that are not anticipated nor intended. Moreover, altering http request may allow communication from P2P module to handler but it does not solve the case where full duplex is required.

The second idea is communication via the database. The p2p module will process and store all information into the database then the handler can query from the database to be displayed on the web page. The advantage is that the interface for handler and database is already built into Yesod, and we have made an interface between the p2p module and the database. However, a major problem occurred when trying to implement the application in this manner. There is no way to construct a reactive application as p2p module has no idea when or if data has been stored from the handler. When information is submitted via http web form, the handler will store the information into database, however, there is no way to indicate to p2p module that new data has been stored

and needs processing. One idea in solving this solution was for p2p media to pool periodically. However, that introduces unreliability into the system which is undesirable.

The third and preferred solution is to create a new process dynamically inside each handler when they are individually called, then pass information via message passing to the p2p module. Since processes in Cloud Haskell utilize light-weight threads, it does not add much overhead to the system. However, a problem arises when trying to use the same node from the p2p module to spawn a new process from the handler. The main stumbling block is that components inside a monad restrict usage from outside of the monad. Therefore, since the node was created inside the p2p module, it cannot be used by outside components, in this case by handlers. As a work around, we decided to create a new node explicitly for use in creating processes for handlers. This creation of a new node does not add complications to the overall system as communication between processes are the same, no matter if they reside on the same node or not. However, conceptually it may not be the best solution since it does not make sense for one machine to have two nodes. But then again we can abstract this complexity out and call it a peer, where a peer contains two nodes.

B. Peer Connection[8]

The algorithm used for peer connection in our application is straight forward. We decided not to use a more complex algorithm in implementing the connection since it was not the focus of this paper. The algorithm adopted will produce an interconnecting web of nodes; all individual nodes will be connected with every other node. This technique may not be the most efficient but it is simple and workable for the application at this stage. In the future we would like to implement better algorithms such as the Chord algorithm.

For a node in a cloud attempting to connect to a remote node in a separate cloud, the algorithm used for connection is as follows[6].

1. If evaluation queue is not empty, evaluate the first remote node in the evaluation queue.
2. Connect to the remote node and send information re all known nodes in current residing cloud.
3. Receive confirmation from remote node which includes information about known nodes of the remote node.
4. Put monitor on remote node and put received node information into evaluation queue that is not already known.
5. Execute step 1.

C. Tweeting Service

The tweeting service would be a global messaging system where anyone that sends a “tweet” would be seen by any other person in the cloud. The tweets would be displayed as a stream of messages something like the wall in Facebook. This is a primary service that allows people to connect and interact. The idea is that people would share useful information which is beneficial for all members of the cloud, thus growing the community as a whole.

While developing the service we found it clear and easy to implement and reason with the code. Some bugs occurred when building this component. However, as functions are contained and components are in logical order, it was fairly easy to find and debug the service.

The algorithm is as follows[6].

1. User inputs message in html form and is submitted to the handler handling that web page.
2. The handler sends the message to the Controller of p2p module via message passing.
3. Once Controller receives the message it will then broadcast the message to all Listeners (residing in p2p module), including its own.
4. Once Listener receives the message, it will store the data into database.
5. Message will be displayed when a web page is loaded.

During the implementation of this service, we encountered the first hurdle that the exotic nature of Haskell, more specifically Yesod, poses. Our intuition is that, when a message is posted the message should automatically be loaded onto the web page AJAX style. However, when searching for suitable JavaScript library to accommodate this feature none was directly compatible. Many libraries require putting some code in the backend in order for it to function, but suffice it to say none supported Haskell. We view this as a minor detail as it was anticipated that not all web tools will currently be fully available to the Haskell web developer. However, on general we found that most web tools that do not explicitly require backend interaction are fully compatible with Yesod.

D. Article Service

The article service provides a platform for users to create rich text articles and share it with other members in the cloud. Users can create or search for articles within the cloud community, thus providing another interesting and exciting way to share valuable information with others.

The original design for the article service was to add redundancy so that articles can be reconstructed. This is a technique where members of the cloud would cache a small percentage of all articles. Thus, as long as a percentage (e.g. 80%) of nodes is on-line then all articles would be able to be reconstructed and be available for sharing. However, after analyzing on how to accomplish such a feat, we quickly realized that this may be acting against the fundamental principles of actor model and purity as it introduces a shared state between peers. If the threshold of percentage needed to reconstruct all articles is not met, then it might render the whole system unusable.

After careful consideration and inspiration from actor model, we decided that the articles created should be store in the individual peer database only. In order to share articles, we will send articles to query nodes on demand. This architecture will reduce duplication and concurrency issues, and also allow the system to be more robust and stable. With this

implementation, we allow nodes to fail whilst not affecting the system as a whole. The tradeoff, however, is that the availability of content would be sacrificed. Nevertheless, we feel this is a fair trade-off, taking into the account that that when querying articles, users mostly make fuzzy searches, thus the lack of nodes would not be felt since users do not know that the failed node and its information was there in the first place. This just means that the more nodes there are in the cloud system, the more likely you will be able to find the desired content, therefore, creating this reliance on each other building a sense of camaraderie amongst peers. This furthers the tight bonding that the application is intended to create.

On the design of how to create and query articles, we choose to use CKEditor to create rich content. The free open source html text editor is fully compatible with Yesod which helps to give the user more tools in creating useful content. The article is stored locally as stated before, and will be queried when needed. The query algorithm is as followed[6].

1. User inputs query in html form and is submitted to the handler handling that web page
2. The handler sends the message to the Controller of p2p module via message passing.
3. Once Controller receives the message it will then broadcast the message to all Listeners (residing in p2p module), including its own.
4. Once Listeners receives the message, it will query its database then return a list of articles to the original query Controller.
5. Query Controller gathers all articles then it sends all articles back to the handler.
6. Handler sends http response to web browser.

E. Web User Experience

Designing and implementing the web interface was challenging. It requires a whole new set of tools and skills to accomplish an outstanding experience. When building the interface, we were pleasantly surprised at how compatible Yesod was with normal web designing tools. The Shakespeare templates made working with the tedious and messy HTML, CSS and JavaScript clear and logical. Many readily available tutorials or examples found on the web can easily be incorporated with minor modification. Even AJAX web page loading was easily implemented with loading icons and other user-friendly designs.

VI. Conclusion

Building a distributed system is not a trivial task. Being able to give guarantees and assurance that the system will function correctly at all times under all circumstances is an immense challenge. As shown in this paper, we believe the better way of building a distributed system is to adopt strict disciplines namely actor model and purity when constructing the application. To our knowledge Cloud Haskell is the paramount tool that distinctly puts prominence on these values. We built a real world application to demonstrate that not only is it capable of delivering but also able to deliver it well. In the process of building, we found that it was easy to reason and understand the code and that most bugs was caught during compile time. Furthermore, when some bugs did slip past the compiler, we easily corrected the error due to the small percentage of code that could cause the error. Library support was wide-ranging, and searching for specific functionality from a specific package was made extremely easy by the centralization of packages on HackageDB[2]. To conclude, we feel that Haskell is an outstanding language for developing real world distributed systems, and that it should be adopted by enthusiastic programmers who want to find a better way of producing high quality software.

VII. Future work

We feel the key to success of this P2PMedia application is to have rich content and high availability when required. This means that we need to find a way to allow users to want to keep the application running so that others can obtain valuable information from it. Thus we feel that moving towards smart phones and mobile devices is an unavoidable direction that needs to be taken. As observed, people generally do not turn off their smart phones as they have become almost an extension of one's being. Therefore, moving towards mobile devices will be important to the success of the application. We have done some research into the feasibility of achieving this ambition, and found several groups of developers working towards making Haskell run on mobile devices.

In regards to the application itself, we would like to add additional features so that there are more ways of interacting with each other in the cloud. We feel that the right architecture is already in place for future expansion. As to the peer to peer algorithm, we are planning to modify the current algorithm to Chord algorithm due to the fact that it has been proven to show better performance, efficiency and scalability.

Source https://bitbucket.org/coolhhc/p2pmedia-yesod1.2

References

[1] 1 million is so 2011. http://blog.whatsapp.com/index.php/2012/01/1-million-is-so-2011/.

[2] Hackagedb. http://hackage.haskell.org/packages/hackage.html.

[3] Hayoo. http://holumbus.fh-wedel.de/hayoo/hayoo.html.

[4] Hoogle. http://www.haskell.org/hoogle/.

[5] Moore's law. http://en.wikipedia.org/wiki/Moore's_law.

[6] P2pmedia powerpoint presentation. https://bitbucket.org/coolhhc/p2pmedia-yesod1.2/downloads/P2PMedia-Presentation.pptx.

[7] G. Agha. Actors: A model of concurrent computation in distributed systems, 1986.

[8] Alexander Bondarenko. Control-distributed-backend-p2p. http://hackage.haskell.org/packages/archive/distributed-process-p2p/0.1.0.0/doc/html/Control-Distributed-Backend-P2P.html.

[9] Francesco Cesarini and Simon Thompson. Erlang programming. gravenstein highway north, sebastopol, oâĂŹreilly media., 2009.

[10] Sadek Drobi. The way github helped erlang and the way erlang helped github. http://www.infoq.com/interviews/erlang-and-github.

[11] Frank Elassen. Introduction to distributed system. http://www.uio.no/studier/emner/matnat/ifi/INF5040/h07/undervisningsmateriale/INF5040_WK35_IntroDS.pdf.

[12] Jeff Epstein, Andrew Black, and Simon Peyton-Jones. Towards haskell in the cloud, 2011. In Proceeding sof the Haskell Symposium.

[13] Haskellwiki. Functional programming. http://www.haskell.org/haskellwiki/Functional_programming#Purity.

[14] C. Hewitt. A universal modular actor formalism for artificial intelligence, 1973.

[15] John Hughes, 1990.

[16] Jussi Kangasharju. Distributed systems: What is a distributed system? http://www.cs.helsinki.fi/u/jakangas/Teaching/DistSys/DistSys-08f-1.pdf.

[17] Merriam-Webster online English dictionary. http://www.merriam-webster.com/dictionary/indeterminism.

[18] Bryan OâĂŹSullivan, John Goerzen, and Don Stewart. Real world haskell.

[19] David Reiss. Thrift: (slightly more than) one year later. https://www.facebook.com/note.php?note_id=16787213919&id=9445547199&index=.

[20] Romain Slootmaekers. The game of distributed systems programming. which level are you? http://blog.incubaid.com/2012/03/28/the-game-of-distributed-systems-programming-which-level-are-you/.

[21] Ruben Vermeersch. Concurrency in erlang and scala: The actor model. http://savanne.be/articles/concurrency-in-erlang-scala/#ref5.

[22] Jim Waldo, Geoff Wyant, Ann Wollrath, and Sam Kendall. A note on distributed computing. http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.41.7628.

[23] Yogsototh. Haskell web programming. http://yannesposito.com/Scratch/en/blog/Yesod-tutorial-for-newbies/#fnref1.