The Report Committee for Neel Harish Shah
Certifies that this is the approved version of the following report:

Generic System Architecture for Context-Aware, Distributed Recommendation

APPROVED BY
SUPERVISING COMMITTEE:

Supervisor:

____
Christine Julien

____
Suzanne Barber

Generic System Architecture for Context-Aware, Distributed Recommendation

by

Neel Harish Shah, B.S.E.E.

Report

Presented to the Faculty of the Graduate School of

The University of Texas at Austin

in Partial Fulfillment

of the Requirements

for the Degree of

Master of Science in Engineering

The University of Texas at Austin

December 2016

Dedication

This work is dedicated to my friends and family who have always given me hope and supported me throughout this voyage.

Acknowledgements

This work would not have been possible without the copious support of my parents, Harish and Rita. Also, I would like to thank my classmates Sonia Marginean and Asim Saleem for encouraging me to pursue my ideas.

Abstract

Generic System Architecture for Context-Aware, Distributed Recommendation

Neel Harish Shah, M.S.E.

The University of Texas at Austin, 2016

Supervisor: Christine Julien

In the existing literature on recommender systems, it is difficult to find an architecture for large-scale implementation. Often, the architectures proposed in papers are specific to an algorithm implementation or a domain. Thus, there is no clear architectural starting point for a new recommender system. This paper presents an architecture blueprint for a context-aware recommender system that provides scalability, availability, and security for its users. The architecture also contributes the dynamic ability to switch between single-device (offline), client-server (online), and fully distributed implementations. From this blueprint, a new recommender system could be built with minimal design and implementation effort regardless of the application.

Table of Contents

List of Tables viii

List of Figures ix

INTRODUCTION 1

LITERATURE REVIEW 4

APPROACH 6

Overview 6

Contributions 8

Anonymous User Profile 8

Group Profile 8

Availability 9

IMPLEMENTATION 11

ANALYSIS 12

Scalability 12

Security 13

Availability 16

FUTURE WORK 19

CONCLUSION 20

Appendix 21

References 28

Vita 30

List of Tables

Table 1: Architecture Component Overview 7

List of Figures

Figure 1: Component Organization and Communication Paths 7

Figure 2: User Group Initialization Performance 13

Figure 3: User Profile Security Tests...15

Figure 4: Profile Server Availability Test..17

Figure 5: Session Server Availability Test 18

Figure 6: User Profile Subsets 22

INTRODUCTION

As Web 2.0 technology matures, recommendations are becoming a highly valued feature for websites and applications. This has driven interest in researching recommendation algorithms and methodologies of varying complexities and domain-specific optimizations. In general, recommender systems have two first-class entities: users and items. Recommender systems generally utilize user preferences and item relationships to provide recommendations.

Context-aware recommender systems, which allow recommender systems to use other applicable information besides user preferences to make recommendations, have become increasingly relevant to the area. This is because it has been shown, in many domains, that it is not sufficient to rely only on the users and items in the system to make accurate recommendations; the context in which a user takes an action toward an item is directly relevant to how another user might behave in a similar context [1]. For example, a user's interests might be affected by the current date, season, or temperature. Thus, it is important to capture these data when a user records their preferences so that recommendations to other users in a similar context are improved.

Although context-awareness leads to better recommendations, it introduces several issues. Managing context information increases the size of the data managed by the system, creating a larger workload and emphasizing the need for a scalable recommender system. Portability of a mobile recommender system could be affected due to the unreliable nature of context information. Context information also poses a new security problem because of the additional sensitive data now managed by the system [2].

The primary requirement of a recommender system is two-fold: to understand the preferences of users for items, and to use these preferences to make recommendations for items to other users of the system [3]. But as research and practical use cases for recommender systems mature, it is clear that there are secondary requirements regarding scalability, availability and security.

Scalability is an issue to be tackled with any system that experiences workload growth. For recommender systems, this generally means an increase in the number of users. Obviously, large-scale recommender system architectures have been designed, deployed and are in use today. Companies like Amazon, TiVo, and others employ these systems to make recommendations to their users [4]. Moreover, the algorithms implemented in these architectures are rather complex due to domain-specific optimizations and deal with an enormous amount of data.

Another aspect of scale for recommender systems is portability: providing recommendations to users that are on a mobile device. Portable recommendations have become relevant only in the last decade, due to the rapidly increasing number of smartphone users [5].

Portable recommender systems have risen to the challenge but pose new problems due to the transient and unpredictable nature of smartphone devices. Additionally, mobile devices emphasize recommendation practices that were not prevalent before, such as context-awareness. Location and device type are good examples of contextual information that is useful when providing recommendations, and there are plenty more.

There are two main security issues with recommender systems: the privacy of personal information and reliability of recommendations [6]. A recommender system can only function if its users are willing to provide the system with their preferences. This information is considered sensitive by the user and should be protected by the system and not used maliciously. Ideally, the users would trust the system to provide them with reliable recommendations. This means the parts of the system that could alter the data used for recommendations need to be secured.

When designing a large-scale software system as a solution to a problem, software engineering best practices dictate considering requirements and developing a reusable, component-based architecture driven by those requirements before implementation [7]. In many cases, an architecture can itself be reused if it is generic enough. The goal of this paper is to propose an architecture blueprint for a context-aware recommender system that has features desired in large-scale implementation.

LITERATURE REVIEW

There have been several architectures proposed for distributed recommender systems. In general, the proposals are limited in their features and are not suited for large-scale implementations. There are a few noted exceptions, and this paper builds on those ideas.

Adomavicius and Tuzhilin discuss the definition and nature of context information in Context-Aware Recommender Systems (CARS) as well as paradigms for using the information for making recommendations [1]. They mention a scalability issue with the dimensionality of the context information but do not discuss storage approaches. Roberts et al. consider context-awareness for recommendations when deriving a system architecture and go into implementation details of data structures for geographic context and associated storage mechanisms, but only for a single-server architecture [5]. This paper will describe the architectural requirements to support distributed context-awareness but does not explore details of context utilization.

In terms of generic approaches, there are various client/server proposals such as the single server architecture proposed by Castagnos and Boyer [8]. In this paper, they propose a hybrid recommendation algorithm that addresses privacy concerns and is implemented on the client/server architecture by keeping sensitive information client side and anonymizing any server side information. A similar approach, called HyRec, is taken by Boutet et al. and shifts computational expense to the client in the name of cost [9]. The two approaches are similar but focus on different goals. However, both ignore certain scalability issues such as node failure and load balancing.

Castagnos and Boyer later addressed scalability issues by adopting a P2P architecture for a decentralized algorithm [10]. They claim to maintain privacy by disassociating user identity through a unique ID, but do not discuss the generation of this ID. Further, they mention identification of malicious users but do not elaborate on that process. They do concede that the decentralized architecture requires more network traffic and reference the PocketLens paper as a source for alternative P2P architectures to reduce this traffic. In this popular paper, Miller et al. consider multiple architectures (centralized and decentralized) on which to implement the PocketLens algorithm [6]. The goal of the personal recommender algorithm implementation was portability and trust for users of the system, which they show can be achieved in a variety of architectures with varying performance and scalability. Of the architectures discussed, each was shown to either have potential security flaws or significant degradation in performance.

A common theme for distributed recommender systems is the security of the sensitive data in the system, particularly if it comes directly from a user. There are algorithmic approaches such as the obfuscation technique Boutet et al. created in [11] which produces a tradeoff between privacy and accuracy. This tradeoff is also shown by Castagnos and Boyer in [8] to a lesser degree when clustering user profiles. There are also more architectural approaches such as the three-layer anonymity model called FreeRec described by Boutet et al. in [12], though this approach did only consider one type of attack. This paper takes an architectural approach to address security and elaborates on components proposed by Castagnos and Boyer in [8] and [10].

In summary, the current literature provides architectures that each focus on a subset of the requirements for a large-scale recommender system. This paper proposes a distributed architecture that accounts for node failure and load balancing. In addition, the architecture adds a layer of security to enable operation in three different modes: online (client-server), distributed, and offline.

APPROACH

This section provides an overview of the proposed architecture and highlights the contributions in this paper to the area. Table 1 explains the four main components: the Context Sensor, the Recommender, the Session Server, and the Profile Server. Figure 1 depicts the organization and communication paths of these components.

Overview

Though this section discusses the logical layer of the architecture, it is necessary to define two component zones that illustrate the mobile nature of the physical deployment layer. The two zones are the Transient Zone and the Static Zone. The Transient Zone contains components that will be deployed on a set of transient nodes: a set of mobile, heterogeneous, and impermanent nodes. In the Static Zone, components will be deployed on a set of static nodes: a set of immobile, relatively homogenous, and permanent nodes.

In general, the Transient Zone can be thought of as containing mobile devices such as phones, tablets and IoT sensors with little resources. Conversely, the Static Zone contains machines with more resources. Since the components in the Static Zone function as servers, the Client/Server organization is emphasized. This is made more apparent in Figure 1. Also in Figure 1, it is shown that Recommender components can talk to each other, which illustrates the distributed property of the Transient Zone. The full nature of this behavior is discussed in the next section but this communication is still possible when connections to the Static Zone are closed. Those connections are managed by Session Server components; this paper entertains the case where no Profile Servers are available or no Session Servers are available. For a full description of all components, please see the Appendix.

Component	Zone	Data	Function
Context Sensor	Transient	Context	Manages Context information used to build Profiles and request Recommendations
Recommender	Transient	User Profile, Group Profile, Recommendations, Subscriber List	Manages Profiles and provides Recommendations to the user
Session Server	Static	Sessions	Manages Session state between the Transient and Static Zone
Profile Server	Static	Anonymous User Profiles	Manages Anonymous User Profiles

Table 1: Architecture Component Overview

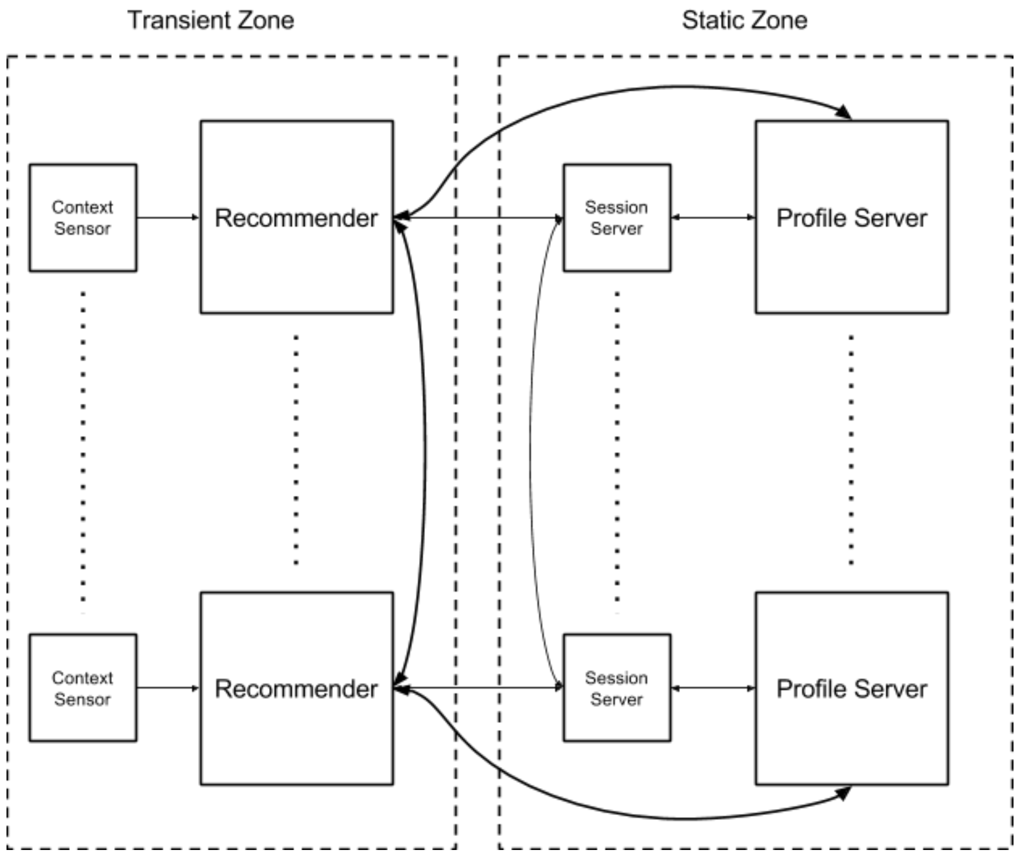

Figure 1: Component Organization and Communication Paths

Contributions

Anonymous User Profile

User Profiles, which are simply a collection of Preferences a User has for Items in the system, can leave the user's device to be served to other users in the client/server architecture. In order to provide privacy to the user, that data must be secured in some way. The most efficient family of methods alters the data so that it is not traceable back to an individual user. Castagnos and Boyer adopted user modeling as an approach for serving secure User Profile data from the server [10]. Their approach clustered similar users together to build typical User Profiles. Security can also be achieved by obfuscating the data in other ways, for example, by adding noise to the profiles [13]. For the purposes of this paper, this process is abstracted and the resulting data is named an Anonymous User Profile.

A User of this recommender system can be asked to categorize their Preferences into three different categories: Public, Private, or Protected. Public information can be accessible to everybody, even entities outside of the recommender system. Private information cannot be accessible to anybody other than the owner of that information, even inside of the recommender system. Protected information can be accessible to everybody, but must not be traceable back to the user. Hence, the Protected User Profile subset is used to generate the Anonymous User Profiles kept on the servers. Note that the Protected User Profile includes the Public User Profile. For a full description of the relationship between the categories, please see the Appendix.

Group Profiles

Group Profiles were proposed by Castagnos and Boyer and represent "a virtual community of interests" distributed across the peers in their system [8]. They proposed to build Group Profiles based on the Public User Profiles of a user’s peers performing above a threshold on a similarity metric. These Group Profiles would then be updated when each Public User Profile was changed. This paper extends the membership of Group Profiles to include Anonymous User Profiles as well as Public User Profiles. This enables the system to perform reasonably well when components in the Static Zone are unavailable. In effect, the system can switch from a client/server implementation to a fully distributed implementation since it now has access to information in other components in the Transient Zone similar to what it was getting from the servers in the Static Zone.

It is worth noting that the Public User Profiles present in a User’s Group Profile have already been deemed similar enough to the User and can be considered good sources for Anonymous User Profiles similar to the current User Profile. However, there may be performance degradation in terms of the accuracy of recommendations since not all Anonymous User Profiles will be available. In particular, there is no guarantee that the best Anonymous User Profile based on the Context may not be available.

AVAILABILITY

As shown in previous work, a recommender system can function in a client-server architecture or in a completely distributed architecture. The proposed architecture offers a marriage of both. If all components are available, recommenders can maintain a Group Profile using a Group Profile Algorithm (Appendix A) that requests Profiles from other Recommenders and Profile Servers (via a Session Server). If Profile Servers become unavailable, the Session Servers can redirect Recommenders to other Profile Servers. If a Session Server cannot provide a Profile Server or it goes down itself, there may be other Session Servers to serve the Recommenders. Alternatively, the Recommenders can continue asking other Recommenders for Profiles (Recommenders can provide Anonymous Profiles from their Group Profile if needed). In this state, the system is completely distributed. On the other hand, if there are no other Recommenders available to ask for Profiles, a Recommender can rely on the Session and Profile Servers for updated information. If no other components are available, the Recommender can fall back to the Group Profile it is maintaining. In the worst case, the Recommender has never built a Group Profile, which might force an Item-Item Recommendation Algorithm (Appendix A) to be used.

IMPLEMENTATION

The implementation was created as a Proof of Concept (POC) and serves as a mechanism to test the architecture design for key features like scalability, privacy and availability. The Scala language was chosen due to the succinctness of syntax and clarity attainable for feature driven testing [14]. This resulted in fewer lines of code for the base implementation and expressiveness of features when writing programmatic test cases.

The Akka message passing framework as written in the Scala language was chosen to implement the POC. Message passing allows implementation of a client-server interaction without restriction to that type of interaction. In addition, the Akka framework provides an abstraction for components called Actors [15]. The Actors' capabilities mesh well with the behaviors of components as designed in this paper. The framework also allows monitoring of Actors in the system, enabling the availability feature of the proposed architecture.

Due to a message-passing framework being chosen, serializable data structures were used to facilitate a quicker implementation. In the Scala language, the case class concept allows for easy creation of serializable data structures. However, due to several layers of abstraction and restrictions of the language, case classes were not used everywhere. Therefore, some of the complex structures like Profiles had to be modeled as map structures [16].

In order for the system to be programmatically tested, some of the algorithms required for the system were given simple implementations. These should not be used for real implementations of such a system and their performance is not indicative of the performance of alternate algorithms at scale.

ANALYSIS

The analysis of the implementation focused on scalability, security, and availability of the architecture. All testing was done on a single physical machine with 4 logical cores and 16 gigabytes of memory. The datasets used for scalability tests were randomly generated Users and Preferences and the other functional tests were smaller, reproducible datasets.

Scalability

A recommender system's performance should scale with the number of Users in the system. Since the Recommendation Algorithm was not implemented practically, it cannot be used to judge the performance of the system. From an architectural perspective, since the Recommendation Algorithm is supposed to run on a single component (the Recommender), we can claim that the architecture does not inherently affect the performance of the Recommendation Algorithm, but rather the physical specification of the device on which the Recommender is deployed.

What we can measure is how long it takes to form a group of a certain number of Users. This is simply the amount of time it takes Recommenders to initialize (gathering Profiles from other Recommenders and Profile Servers in their group). Since the current implementation pushes every update to any Profile to all subscribers (components that have that Profile in their Group Profile), there is a natural polynomial trend as shown in Figure 2. These measurements could help identify a threshold of concurrent Users within a User Group.

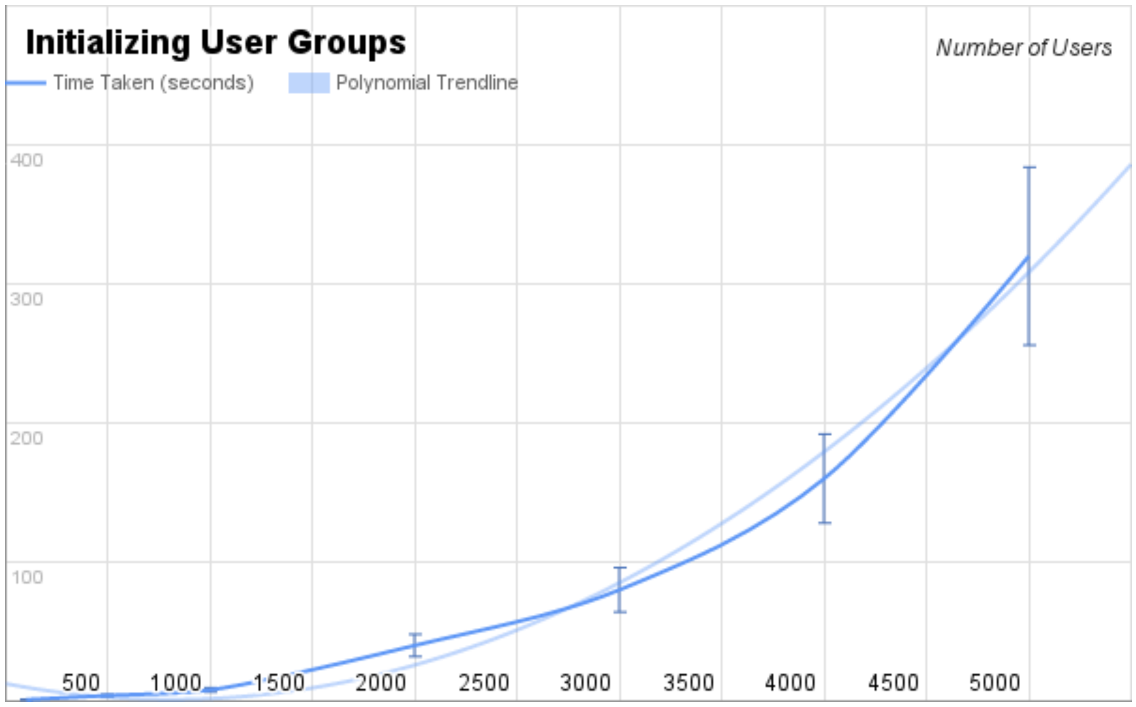

Figure 2: User Group Initialization Performance

Security

There are two requirements of the system for security of user information: privacy and reliability. The proposed architecture does its best to address these requirements directly without relying on underlying algorithms. Of course, security algorithms relating to encryption, gossip, and obfuscation can be applied to the final implementation to address any additional requirements.

From the perspective of privacy, Public User Profiles have no security constraints on them and are the only part of the Profile exposed by a Recommender to another Recommender. Protected User Profiles are only exposed to Profile Servers through a Session granted by a Session Server. The Session Servers manage the Profile Server balancing and so are aware of only legitimate Profile Servers. The scenario tests in Figure

3 show that the User Profile implementation is able to provide appropriate subsets of the Profile. Note that the APIs used in the tests would be inaccessible to external Actors.

Even though the Protected User Profile is sent without any indication of which User it belongs to, an adversary who is posing as a Session Server might still be able to get access to it. This issue is handled by initializing a Recommender with references to trusted Session Servers. Finally, Private User Profiles never leave the Recommender that owns the Profile. This behavior is ensured by the Recommender implementation.

From a reliability standpoint, an adversary posing as a Recommender could also fool the Profile Servers. The false Recommender could inject Preferences into the Anonymous User Profile managed by the Profile Server. This is difficult to prevent but the effect is mitigated since there are many Profile Servers and the Preferences are weighted and merged with others in the Anonymous User Profiles. Additionally, the Recommendation that the system makes is not solely based on Anonymous User Profiles but also other User Profiles in the Group Profile, reducing the effect of a malicious Profile further.

```
feature("User profile should filter preferences based on privacy level") {
 scenario("Asking for private data should return all data") {
  Given("a user profile")

  val preferences: ItemPreferences = mutable.Map(
   (Item(getTestID), None) -> Preference("5"),
   (Item(getTestID), None) -> Preference("4", PROTECTED),
   (Item(getTestID), None) -> Preference("3", PROTECTED),
   (Item(getTestID), None) -> Preference("2", PUBLIC),
   (Item(getTestID), None) -> Preference("1", PUBLIC)
  )

  val userProfile = UserProfile(User(getTestID), preferences)

  When("asking for private data")

  val filteredPreferences = userProfile.privateCopy
```

Figure 3: User Profile Security Tests

```
   Then("the profile should contain the same preferences")

   assert(filteredPreferences.profile.equals(preferences))
 }
 scenario("Asking for protected data should return protected and public data") {

   Given("a user profile")

   val protectedPreferences: ItemPreferences = mutable.Map(
     (Item(getTestID), None) -> Preference("4", PROTECTED),
     (Item(getTestID), None) -> Preference("3", PROTECTED),
     (Item(getTestID), None) -> Preference("2", PUBLIC),
     (Item(getTestID), None) -> Preference("1", PUBLIC)
   )
   val preferences: ItemPreferences = mutable.Map(
     (Item(getTestID), None) -> Preference("5")
   )

   val userProfile = UserProfile(User(getTestID), protectedPreferences ++ preferences)

   When("asking for private data")

   val filteredPreferences = userProfile.protectedCopy

   Then("the profile should contain only the protected preferences")

   assert(filteredPreferences.profile.equals(protectedPreferences))
 }

 scenario("Asking for public data should return only public data") {

   Given("a user profile")

   val publicPreferences: ItemPreferences = mutable.Map(
     (Item(getTestID), None) -> Preference("2", PUBLIC),
     (Item(getTestID), None) -> Preference("1", PUBLIC)
   )
   val preferences: ItemPreferences = mutable.Map(
     (Item(getTestID), None) -> Preference("5"),
     (Item(getTestID), None) -> Preference("4", PROTECTED),
     (Item(getTestID), None) -> Preference("3", PROTECTED)
   )

   val userProfile = UserProfile(User(getTestID), publicPreferences ++ preferences)

   When("asking for private data")
   val filteredPreferences = userProfile.publicCopy

   Then("the profile should contain only the protected preferences")
   assert(filteredPreferences.profile.equals(publicPreferences))
 }
}
```

Figure 3, cont.

Availability

For availability, the proposed system offers a capability to switch between offline, client-server and fully distributed operations. In order to test this fully, components in the system had to be taken out at runtime. The first availability test suite, shown in Figure 4, made sure Session Server components function appropriately if Profile Servers unexpectedly went down. This means the Session Server remained stable (all Session requests were handled) and the Profile Server was not introduced to any Recommenders that requested a Session after the Profile Server went down.

The second availability test suite, shown in Figure 5, ensured that Recommenders function appropriately when Session Servers unexpectedly went down. This means the Recommender remained stable (all Recommendation requests were handled). Additionally, a system initialized with only Recommenders and Context Sensors should remain stable.

```
feature("Session servers should function if profile servers die") {
 scenario("A single profile server dies") {
  Given("a session server managing two profile servers")

  val testContext = Some(Context("testContext"))
  val profileServer = TestActorRef(new ProfileServer(testContext))
  val profileServer2 = TestActorRef(new ProfileServer())
  val sessionServer = TestActorRef(new SessionServer(mutable.Map(
   testContext -> mutable.Set(profileServer),
   None -> mutable.Set(profileServer2)
  )))

  When("one profile server is shut down")

  profileServer ! PoisonPill

  And("we ask for a session for what would have been its context")

  val future = sessionServer ? SessionRequest(testContext)
  val future2 = sessionServer ? SessionRequest()

  Then("it should not be returned")
```

Figure 4: Profile Server Availability Test

```
    val Success(session: Session) = future.value.get
    val Success(session2: Session) = future2.value.get

    session.actorRef shouldEqual None
    session2.actorRef shouldEqual Some(profileServer2)

  }

  scenario("No profile servers left should not be a problem") {
    Given("a session server managing two profile servers")

    val testContext = Some(Context("testContext"))
    val profileServer = TestActorRef(new ProfileServer(testContext))
    val profileServer2 = TestActorRef(new ProfileServer())
    val sessionServer = TestActorRef(new SessionServer(mutable.Map(
      testContext -> mutable.Set(profileServer),
      None -> mutable.Set(profileServer2)
    )))

    When("both profile servers are shut down")

    profileServer ! PoisonPill
    profileServer2 ! PoisonPill

    And("we ask for sessions for what would have been their contexts")

    val future = sessionServer ? SessionRequest(testContext)
    val future2 = sessionServer ? SessionRequest()

    Then("the session server should be able to hand out empty sessions")

    val Success(session: Session) = future.value.get
    val Success(session2: Session) = future2.value.get

    session.actorRef shouldEqual None
    session2.actorRef shouldEqual None
  }
}
```

Figure 4, cont.

```
feature("Recommenders should function with only other recommenders") {
 scenario("No profile servers left should not be a problem") {
   Given("a session server that has no active profile server")
   val sessionServer = TestActorRef(new SessionServer())

   When("a recommender is initialized")

   val recommender = TestActorRef(new Recommender(User(getTestID), mutable.Set(),
mutable.Set(), mutable.Set(sessionServer)))

   Then("the recommender should be able to serve a recommendation")

   recommender ? RecommendationRequest()
 }
}
```

Figure 5: Session Server Availability Test

FUTURE WORK

There are two main areas for future work that are in line with the goal of this paper, architecture design and a reusable implementation. For architecture design, an interesting extension is the User owning multiple Recommender components. This means that the Private User Profile needs to leave the Recommender component that owns it, bringing in a new security problem to solve. For a reusable implementation, there are several things to work on: open-sourcing the code, packaging, documentation, further testing, etc. If the reusable implementation is meant to be large-scale, it could be extended to include integration with Docker, a containerization technology, which would enable that kind of deployment. For example, Amazon Web Services (AWS) offers Docker support and integration.

Other future work might include designing plugin interfaces for the algorithms (for example, the Recommendation Algorithm) and even providing extendable implementations for the algorithms. An interesting algorithm to write would be the Session Algorithm for balancing across Profile Servers. The simple implementation is balancing across different Contexts but there are issues to overcome with that approach.

Finally, more work on the data structures used to represent Profiles, Context and Preferences could make the system even more flexible. For example, the simple implementation of Context is a tuple of categorized values. Introduction of a "fuzzy" Context that is not strict on specific values could benefit the Profile Anonymization Algorithm, which has to consider Context when merging Profiles, and possibly even the Session Algorithm mentioned above.

CONCLUSION

The goal of this paper was to present an architecture blueprint for a context-aware recommender system that provides scalability, availability, and security for its users. Though there is much room for improvement, the analysis reveals that this goal was met. Though the scalability of the architecture depends mostly on the physical layer, it was shown that group formation is polynomial in time. Additionally, balancing and node failures were taken into account, problems that become more prevalent as the system scales. Security issues in the architecture were pushed out of the network with the Session Server handing out trusted Sessions to the Recommender and the Profile Server. Finally, availability of the system is improved over previous works since the absence of components in the Static Zone (servers) are not essential to the desired functionality of the system.

Recommender systems are quickly becoming a necessary part of software products, regardless of industry or application. Providing a architecture blueprint for students and others to use reduces the amount of "boilerplate" design work and coding between projects and introduces a common understanding of what the architecture should be. The base implementation provides a good testing harness for future work and the details in this paper and in the Appendix provide enough information to make improvements without compromising the design goals. Collaborative software engineering across organizational or other boundaries can determine the success or failure of a project, and having common abstractions eases that process.

Appendix

Data

Item

An Item is an entity that can be consumed and rated by or recommended to a user of the system. This paper does not consider where the Items in the system are hosted and served from.

Context

In order to make the recommender system context-aware, the Recommendations must be made after considering the Context of a user. Context is a general term for any information available at the time the Recommendation is requested. For example, the most common Context information is time and location. Knowing the current time and the location of the user could allow the system to provide a better recommendation to the user than if the Context had not been considered.

A Context object has a set of categorized tuples of arbitrary size containing various pieces of information. It is important to note these data are the secondary factor considered when making Recommendations to users of the system, largely due to the fact that they may not always be available.

Preference

Preferences are simply a measurement of a user's affinity towards an Item. The architecture presented in this paper will be agnostic to the actual representation of these measurements and refer to them as Preferences.

Recommendation

Recommendations are simply an ordered set of Items given to a user by the system. This paper does not consider the representation of the Recommendations. They will be generated by a Recommendation Algorithm and are the only output of the system to a user.

User Profile

A User Profile contains entries pertaining to a specific user of the system with regards to Preferences for Items in the system. These entries are the primary factor considered when making recommendations to a user of the system. A User Profile entry is simply a tuple containing an Item or a reference to the Item, a Context object, and a Preference. Since the Context is not always guaranteed to be present, it is considered optional in the entry.

Subsets of the User Profile are allowed to have different semantics: Private, Protected, or Public.

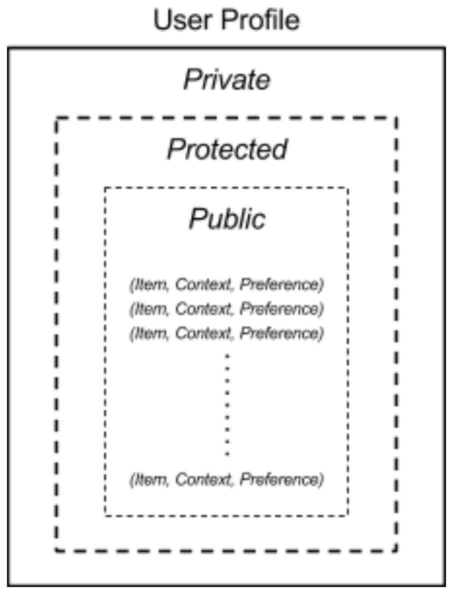
User Profile
Private
Protected
Public
(Item, Context, Preference)
(Item, Context, Preference)
(Item, Context, Preference)
(Item, Context, Preference)

Figure 6: User Profile Subsets

Private

By default, the User Profile will be treated as Private. The Private User Profile will not leave the node that created it without the user's permission and the system cannot use it to make recommendations to other users. However, it can be used to make recommendations to the user who owns it. If the Private User Profile is copied to another node in the system, it first must be encrypted via a Profile Encryption Algorithm.

Protected

A user of the system may elect to allow the system to make recommendations based on their preferences with a caveat of protection of that data. The Protected User Profile can only be present on the node that owns it and sent to static nodes in the system. That is, it cannot be sent to transient nodes in the system. Additionally, it cannot be directly used to make recommendations to other users; it must first go through a Profile Protection Algorithm, which will generate an Anonymous User Model.

Public

A user of the system may elect to allow the system to make recommendations based on their preferences with no restrictions. The Public User Profile can be present on any node in the system and therefore can be directly used to make recommendations to any user of the system.

Anonymous User Profiles

An Anonymous User Profile is a User Profile that is not associated with a particular user and cannot be mapped back to a real User Profile. That is, there is no way to obtain the data in a real User Profile from an Anonymous User Profile. Anonymous User Profiles are generated via Profile Anonymization Algorithm and therefore can be derived from one or more Protected User Profiles. It can be used to make recommendations for any user in the system.

Group Profile

A Group Profile is an entity owned by a Recommender component and is used to make recommendations to a user of the system. Castagnos and Boyer proposed a Group Profile that served as a “virtual community of interests” and contained real User Profiles that passed a threshold based on a similarity metric [10]. This paper extends their idea slightly. Along with real Public User Profiles, Group Profiles can also contain Anonymous User Profiles and will be maintained with a Group Profile Algorithm.

Subscriber List

A Subscriber List is associated with a User Profile and is simply a list of nodes that need to be notified when the User Profile is updated. A Preference algorithm will determine the method and timing of the notification.

COMPONENTS

Context Sensor

The Context Sensor is component contained only within a transient node in the system. As its name suggests, it will manage Context data. Its responsibilities include collecting, storing, updating, and offloading Context data. Context Sensors can communicate with Recommenders, Session Servers, and Profile Servers.

Context Sensors are the only components that produce Context data. This information is then offloaded to Recommenders or Profile Servers. If no instances of those components are reachable then the Context Sensor stores the Context data until an instance is reachable.

Recommender

The Recommender is also a component contained only within a transient node in the system. It will manage data needed to provide recommendations to a user of the system, namely User Profiles and Group Profiles. Its responsibilities include storing, updating, and transmitting Profile data and creating Recommendation objects. The Recommender component can communicate with other Recommenders, Context Sensors, Session Servers, and Profile Servers.

Session Server

The Session Server is a component contained only within a static node in the system. It will manage data related to Sessions between Transient and Static nodes in the system. Its responsibilities include creating and managing Sessions, storing and offloading Session data. Session Servers can communicate with other Session Servers, Context Sensors, Recommenders, and Profile Servers.

Profile Server

The Profile Server is a component contained only within a static node in the system. It will manage data needed to provide better recommendations to a user, namely Anonymous User Profiles. Its responsibilities include collecting, storing, updating, and transmitting Anonymous User Profiles. Profile Servers can communicate with each other and all other components.

ALGORITHMS

This section describes the requirements and provides example implementations for the algorithms that would be implemented in a recommender system adopting the proposed architecture. It is worth noting that this paper proposes a generic architecture that is not implementation dependent and so any implementation examples provided are not required ones.

Recommendation Algorithm

There are many Recommendation Algorithms in existence but this paper will simply list the basic requirements for such an algorithm. A Recommendation Algorithm must take as input the preferences of users of the system and, optionally, associated Context information. It should provide a set of Recommendations as output. In general, there are two parts to the algorithm: finding the best neighbors and choosing which Items belong in the result set.

Profile Anonymization Algorithm

A Profile Anonymization Algorithm should take as input a set of User Profiles and output an Anonymous User Profile that cannot be mapped back to the original User Profiles. It should attempt to preserve the value of the data in the for the system. An implementation that produces an Anonymous User Profile by completely randomizing the data in a User Profile no longer has value since the system cannot make useful Recommendations from it.

Profile Encryption Algorithm

A Profile Encryption Algorithm simply encrypts a User Profile so that it cannot be used maliciously if it is intercepted while being transmitted. The User Profile should be able to be decrypted on the destination, which suggests an asymmetric encryption scheme or possibly an additional optional component.

Group Profile Algorithm

A Group Profile Algorithm manages the Group Profile data in a Recommender component. The algorithm should add relevant User Profiles to the Group and remove and that have become irrelevant. A threshold based on a similarity metric should determine relevancy.

Preference Algorithm

A Preference Algorithm should be able to update the User Profile with an entry using a Preference, Item and optionally a Context object. An additional requirement is that it notify all subscribers to the profile, i.e., the nodes in the Subscriber List.

Session Algorithm

A Session Algorithm is responsible for managing the lifecycle of a Session. This paper defines Session as temporary data interchange between a transient and static node. The algorithm should be able to start and end a Session between a node in the Transient Zone and a node in the Static Zone.

References

[1] G. Adomavicius and A.Tuzhilin. "Context-Aware Recommender Systems," [Online]. Available: http://ids.csom.umn.edu/faculty/gedas/NSFCareer/CARS-chapter-2010.pdf

[2] A. Jeckmans, A. Peter and P. Hartel. "Efficient Privacy-Enhanced Familiarity-Based Recommender System," [Online]. Available: https://pdfs.semanticscholar.org/9a24/f138cc1162e4c24cdafddd542f575066f83d.pdf

[3] G. Adomavicius and A. Tuzhilin. "Toward the Next Generation of Recommender Systems: A Survey of the State-of-the-Art and Possible Extensions," [Online]. Available: http://pages.stern.nyu.edu/~atuzhili/pdf/TKDE-Paper-as-Printed.pdf

[4] T. Z., Z. K., J. L, M. M., J. R. W. and Yi-Cheng Zhang. "Solving the apparent diversity-accuracy dilemma of recommender systems," [Online]. Available: https://arxiv.org/pdf/0808.2670.pdf

[5] M. R., N. D., B. B., K. P., B. P., V. B., A. W. and Paul Rasmussen. "Scalable Architecture for Context-Aware Activity-Detecting Mobile Recommendation Systems," [Online]. Available: https://www.parc.com/content/attachments/scalable-architecture-context-aware.pdf

[6] B. N. M., J. A. K., and John Riedl. "PocketLens: Toward a personal recommender system," [Online]. Available: https://www.researchgate.net/publication/220515913_PocketLens_Toward_a_personal_recommender_system

[7] D. Batory and S. O'Malley. "The Design and Implementation of Hierarchical Software Systems With Reusable Components," [Online]. Available: http://www.cse.msu.edu/~cse870/Materials/Frameworks/tosem-92.pdf

[8] S. Castagnos and A. Boyer. "A Client/Server User-Based Collaborative Filtering," [Online]. Available: https://hal.archives-ouvertes.fr/inria-00104863/document

[9] A. B., D. F., R. G., A. K. and Rhicheek Patra. "HyRec: Leveraging Browsers for Scalable Recommenders," [Online]. Available: https://hal.inria.fr/hal-01080016/document

[10] S. Castagnos and A. Boyer. "Modeling Preferences in a Distributed Recommender System," [Online]. Available: https://hal.archives-ouvertes.fr/inria-00171802/ document

[11] A. B., A. K., D. F., R. G. and Arnaud J´egou. "Privacy-Preserving Distributed Collaborative Filtering," [Online]. Available: https://hal.inria.fr/hal-00799209/file/RR-8253.pdf

[12] A. B., D. F., A. J., A. K. and H. B. Ribeiro. “FreeRec: an Anonymous and Distributed Personalization Architecture,” [Online]. Available: https://hal.inria.fr/hal-00844813/file/main.pdf

[13] F. McSherry and I. Mironov. “Differentially Private Recommender Systems,” [Online]. Available: https://www.microsoft.com/en-us/research/wp-content/uploads/2009/06/NetflixPrivacy.pdf

[14] M.O., P.A., V.C., I.D., G.D., B.E., S.M., S.M., N.M., M.S., E.S., L.S., and Matthias Zenger. “An Overview of the Scala Programming Language,” [Online]. Available: http://www.scala-lang.org/docu/files/ScalaOverview.pdf

[15] S. Tasharofi, P. Dinges and R. Johnson. “Why Do Scala Developers Mix the Actor Model with Other Concurrency Models?,” [Online]. Available: https://www.ideals.illinois.edu/bitstream/handle/2142/34816/fase2013_submission.pdf

[16] A. Miele, E. Quintarelli and L. Tanca. “A methodology for preference-based personalization of contextual data,” [Online]. Available: https://openproceedings.org/2009/conf/edbt/MieleQT09.pdf

Vita

Neel Harish Shah is a Software Engineer and Entrepreneur that graduated from the University of Texas at Austin with a BS in Electrical and Computer Engineering with a Business Foundations Certificate in the Spring of 2013 and a MS in Software Engineering in the Fall of 2016. He currently works for a marketing technology company called MaxPoint and is co-founder and CTO of Zateo Inc.

Permanent email: neel.shah.528@gmail.com

This report was typed by Neel Harish Shah