Identifying Users in the Bridging Service Between Two Different Chat Services Using User Icons

Ko Miyazaki[1](✉) and Haruaki Tamada[2](✉)

[1] Division of Frontier Informatics, Graduate School of Kyoto Sangyo University, Motoyama, Kamigamo, Kita-ku, Kyoto, Kyoto Prefecture, Japan
i1888123@cc.kyoto-su.ac.jp

[2] Faculty of Information Science and Engineering, Kyoto Sangyo University, Motoyama, Kamigamo, Kita-ku, Kyoto, Kyoto Prefecture, Japan

Abstract. There are many chat services in the world, such as Slack (https://slack.com/), Skype (https://www.skype.com/en/), gitter.im (https://gitter.im/), etc. Generally, we cannot send messages over the different chat services, since there are no route to send messages between them. To solve the problem, we propose the bridge system, named *CiBridge*, to exchange messages between different two chat services. By using the *CiBridge*, users in each chat service can send messages to other chat service by using the ordinary chat service. However, one problem arises in use of *CiBridge*. The problem is on the bridged messages which are posted from another chat service. The bridged messages are posted by *CiBot*, therefore, the original user of the message are concealed. Of course, the body of bridged messages shows the user names of original messages. However, the text information does not clarify the original user of the messages. Generally, the users distinguish each user by their avatar icons rather than the user names. For example, GitHub (https://github.com/) supports the user distinguishes each developer by the user icon. If the user does not specify his/her own user icon, GitHub gives the default user icon by Identicon (https://blog.github.com/2013-08-14-identicons/). That is, the visualization strongly helps the instinctive understandings. Therefore, the user icons are important information in the chat system to distinguish each user. This paper tries for embedding the original user icons to the bridged messages.

1 Introduction

Today, the development teams usually use some chat services to exchange messages among the members. For example, Oracle corporation employs Slack as a chat service for their daily use[1]. In such teams, a bot in the chat service solves simple but bothersome works, e.g., reserving the meeting rooms, automatically

[1] https://slack.com/enterprise.

V. G. Duffy (Ed.): HCII 2019, LNCS 11582, pp. 380–390, 2019.
https://doi.org/10.1007/978-3-030-22219-2_29

deployment, and so on. It is called ChatOps to solve the work by the bot like above. For instance, Netflix manages the incidents on their chat service[2].

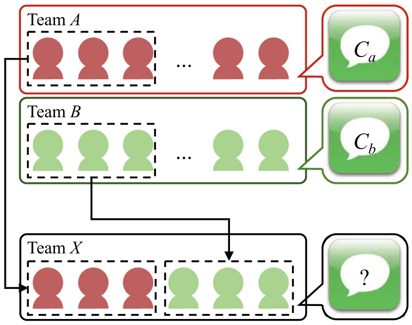


Fig. 1. Issues of the paper

However, the problems arise in the case of collaborating members over the teams. Let consider the case of building the new team X from existing team A and B, shown in Fig. 1. Besides, the members in X remains in the former teams. Additionally, the team A and B employ another chat services C_a and C_b, respectively. In the case, the members of X have to use two or more different chat services. Because, they need to use the chat services corresponding to the members in X, A, B and switching the chat services requires a slight overhead. Of course, the overhead of this case is quite low; however, the overhead gradually gains by joining many teams.

The following four items are the categories of cost by using several chat services.

(1) the searching cost for contact subjects,
(2) the switching cost of chat service.
(3) the registration cost, and
(4) the login cost.

In (1), by using many chat services, the user must remember the contact address and corresponding chat service. This searching cost is generally concealed, however quite high. In (2), the user pays some cost of switching the chat services. The one switching cost is quite slight; however, the cost gradually increases in proportion to the switching count. Next (3), the user should conduct user registration to use the new chat service. This registration cost is mandatory once at first. However, some recent services do not allow multiple accounts; therefore, the user must register cell phone numbers. This restriction is quite bothersome for the user. Finally, in (4), the user tries to communicate with a particular chat service, then the chat service requires authentication. The authentication is generally quite significant; however, this case is also bothersome.

[2] https://www.usenix.org/conference/srecon16/program/presentation/tobey.

To solve the above problems, we propose the bridging service between two chat services, called *CiBridge*. *CiBridge* installs the bot, called *CiBot* into the target chat services. Then, *CiBot* reacts to a posted message from the user, and send it to another *CiBot*. *CiBridge* manages the pair of *CiBot*s. Also, `sameroom.io`[3] is the one of bridging chat service. However, the icon of the posted messages through `sameroom.io` is default user icons, since the bot conducts the post. Generally, the users distinguish each user by their avatar icons rather than the usernames [1,2]. Therefore, `sameroom.io` solves problems by bridging chat services, however, it causes another problem. Therefore, *CiBridge* supports avatar icons in the messages posted by *CiBot*s.

The remainder of the paper shows the proposed method (Sect. 2). Next, it describes the implementation of the proposed method (Sect. 3). Afterward, Sect. 4 shows the case studies of the proposed method. Finally, Sect. 5 discusses the related works, and then, we summarize the paper in Sect. 6.

2 The Proposed Method

2.1 The Chat Services

In the paper, we define the chat services as follows. Figure 2 shows the class diagram of a chat service. From the Fig. 2, the message has a user who post it.

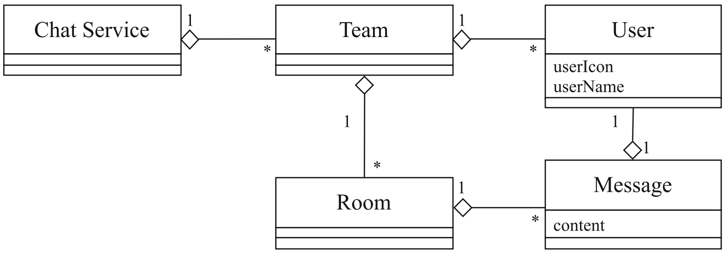


Fig. 2. The class diagram of the chat services for the paper

For illustrating a practical chat service, we choose Slack and Skype. Slack and Skype are popular chat services. The Slack has many teams and organizes users by the teams. Then, a team has several rooms, users in a room joined freely, and the user has conversations on each room.

On the other hand, Skype has only one team, and the team has all of the users in the world. The user managing the room can invite other users to join the room.

In both chat service, the content of a message is the same in the definition of this paper. The message has a posted user and content. Besides, the content of

[3] https://sameroom.io/.

the message is typically string format. However, the content might be a binary format, such as images, videos, audios, and other formats. The proposed method allows any format.

In almost chat service, the users can specify own icons. The user icon is affected to distinguish other users [1–3]. Therefore, the user icon is quite significant even in the bridge services.

2.2 The Proposed Method

To bridge the two chat service, we prepare the one relay server and two chatbots. Figure 3 illustrates the proposed method. At first, we install chatbots into both chat services. The chatbots resident on a particular room, and react the post from the user on the room. The user posts a message (Fig. 3a); then the chatbots send the content of the message to the relay server (Fig. 3b). When the relay server receives the message, it finds corresponding chatbot (Fig. 3c) from the database. Next, the relay server sends the content of the message to the found chatbot (Fig. 3d). Finally, the chatbot receives the message from the relay server, and it posts the message instead of the original user (Fig. 3e).

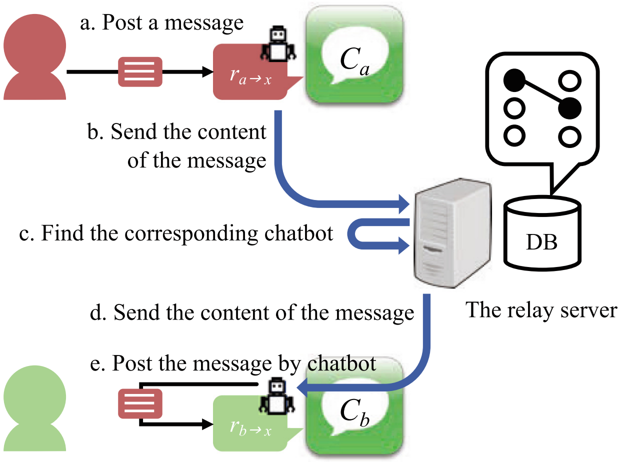

a. Post a message
$r_{a\to x}$
C_a
b. Send the content of the message
c. Find the corresponding chatbot
DB
The relay server
d. Send the content of the message
e. Post the message by chatbot
$r_{b\to x}$
C_b

Fig. 3. The procedures of the proposed method.

The received data on the relay server contains the message itself, and chatbot and posted user information. The relay server manages the pairs of chatbots using the database. Each chatbot is identified by their ids. Also, the user is distinguished by the URL of the team, and the user id. Besides, the user would identify each user by icons, therefore, this paper assumes that almost users set their icons.

2.3 Formulation of the Proposed Method

This section gives the formulation of the proposed method. Let be two teams X and Y, and each team uses different chat services C_x and C_y. Also, the members of each team are shown as $m(X) = \{x_1, x_2, ..., x_{n_x}\}$, and $m(Y) = \{y_1, y_2, ..., y_{n_y}\}$. The new team Z are built by arbitrarily selecting the members of X and Y ($m(X) \cap m(Y) \neq \phi$). The both chat services C_x and C_y introduce the new rooms $r_{x \rightarrow y}$ and $r_{y \rightarrow x}$, respectively, for connecting both rooms. The messages posted on $r_{x \rightarrow y}$ are automatically share to $r_{y \rightarrow x}$ by the relay server B_r, and vice versa.

For this, we install the bridging system *CiBridge* for connecting C_x and C_y through $r_{x \rightarrow y}$ and $r_{y \rightarrow x}$. *CiBridge* composes of several chatbots $b_i \in B_b$ and one relay server B_r. For connecting $r_{x \rightarrow y}$ and $r_{y \rightarrow x}$, a user $x_i \in m(X)$ setups C_x ($1 \leq i \leq n_x$). x_i selects the suitable chatbot b_x from a chatbots set B_b of *CiBridge*. The API of each chat service is generally different. The suitable chatbot means that the bot uses the API of a specific chat service. Then, x_i installs b_x to C_x by authorizing to react the message post and to post messages to $r_{x \rightarrow y}$. The installed chatbot represents as $\beta_{x \rightarrow y}$ The each β_i is identified by its id ($id(\beta_i)$). The relay server B_r manages the relationship between $\beta_{x \rightarrow y}$ and $\beta_{y \rightarrow x}$. That is, if a chatbot is specified, B_r can find the corresponding chatbot. Similarly, $y_i \in m(Y)$ setups C_y for installing b_y, the messages posted on $r_{x \rightarrow y}$ are automatically transferred to $r_{y \rightarrow x}$, and vice versa.

Next, we formulate the messages on the *CiBridge*. When a user x_k posts a message e on a room $r_{x \rightarrow y}$, a chatbot $\beta_{x \rightarrow y}$ activates for bridging the message. The message e contains the message body o and information of the user x_k, the room $r_{x \rightarrow y}$ and the chatbot $\beta_{x \rightarrow y}$ ($e = \{o, x_k, r_{x \rightarrow y}, \beta_{x \rightarrow y}\}$). The message body o is typically plain text message. The information of x_k is the name of x_k and icon.

3 Implementation

3.1 Overview

For the proposed method, we chose Slack and Rocket.Chat[4] as the target chat services C_x and C_y. Ideally, *CiBridge* should support every chat services. However, each chat service has generally different API, and sometimes it is extensively different. Therefore, we develop the chatbot for each chat service step by steps. To support the above two chat services is the first step of the proposed method.

We implemented the relay server B_r as a REST service [4] written in the Java language. Also, we applied Hubot[5] for implementing *CiBot*s. B_r and each *CiBot* run at each Amazon EC2 server on AWS[6]. We used the following libraries for the B_r and *CiBot*s.

[4] https://rocket.chat/.
[5] https://hubot.github.com.
[6] https://aws.amazon.com.

Table 1. The endpoints of B_r

Endpoints	HTTP method	Description
`/api/relay/messages`	`POST`	Exchange posted messages to corresponding chatbot
`/api/relay/pairs`	`GET`	Returns the corresponding chatbot with given room id
	`POST`	Register the room pair for bridging the messages
	`DELETE`	Delete the room pair

- The relay server B_r
  - Java 10
  - Jersey 2.0.1
  - Jetty 9.4.11
  - SQLite 3.27.1[7]
- *CiBot*(Slack)
  - Hubot
  - hubot-slack adapter 4.6.0[8]
- *CiBot*(Rocket.Chat)
  - Hubot
  - hubot-rocketchat adapter 1.0.12[9]

3.2 The Relay Server B_r

B_r was built as a REST service for message bridging. B_r has two endpoints, `/api/relay/messages` and `/api/relay/pairs`. Table 1 shows the available HTTP methods for the endpoints. Besides, the format of the message e is JSON, and the authorization token of *CiBridge* includes in the HTTP header. B_r performs two features, relaying the message, and managing the pairs.

3.2.1 Relaying the Message

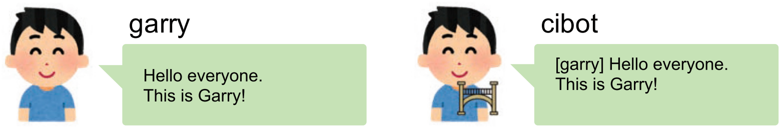
At first, it explains the relaying the message. The relaying the message feature is activated by receiving the message through `/api/relay/messages` by `POST` method, and performed the following stages.

1. extracts the room id $id(\beta_{x \to y})$ from the received message e.
2. finds the corresponding chatbot $\beta_{y \to x}$ by the extracted chatbot id $id(\beta_{x \to y})$.
3. converts the messages e to clarify the bridged message e' ($e' = \{o', x'_k, r_{x \to y}, \beta_{x \to y}\}$).
4. sends the converted message e' to the corresponding chatbot $\beta_{y \to x}$.

[7] https://www.sqlite.org/index.html.
[8] https://github.com/slackapi/hubot-slack.
[9] https://github.com/RocketChat/hubot-rocketchat.

The message e contains the message body o, information of user x, room r_i and the chatbot β_i, described above ($e = \{o, x, r_i, \beta_i\}$). In *CiBridge*, the B_r converts the e to e' by the following two steps. The first step converts the user icon by putting *CiBridge* icon at the lower right of it. The second step adds the original user name at the first of the message body, if the message body is plain text.

Figure 4 shows an example of the original message e and the bridged message e'. This example shows that the user Garry posts a message on the room `HCII2019`, and we bridge between `HCII2019` and the room `PAPERS` in another chat service. Figure 4(a) is the view in the `HCII2019`, and (b) is the view in the `PAPERS`. The *CiBridge* adds its icon to the original user icon, appends the room name `HCII2019` the username, and updates the message body to clarify the bridged message.


garry

Hello everyone.
This is Garry!

cibot

[garry] Hello everyone.
This is Garry!

(a) The original message e (b) The converted message e'

Fig. 4. An example of the original and the bridged messages

3.2.2 Managing the Chatbot Pairs

To manage the chatbot pairs should perform CRUD tasks (create, read, update, delete). However, we substitute the update task for the delete and create tasks. Therefore, the endpoint of `/api/relay/pairs` accepts `GET`, `POST`, and `DELETE` methods, and do not support `UPDATE` method. Similarly with the general REST application, `GET` method returns the list of the registered pairs, `POST` method register the posted pair, and `DELETE` method remove the pair from the list.

In the create task, B_r inserts the pair of chatbots into the database. The pair is specified in the request body of the HTTP request. The inserted data is the chatbot id $id(\beta_i)$, its name, the corresponding chatbot id $id(\beta_j)$ and its url $url(\beta_j)$.

Next, the read task finds the corresponding chatbots for requested $id(\beta_i)$. The task was done for simply reading the database. Finally, the delete task is also executed by the deleting the database entries.

3.3 *CiBots*

CiBot is a simple chatbot built with Hubot. *CiBot* has three features: detecting the new post, posting message instead of the original user, and presenting various information replying the request from a user. In the detecting the new post, *CiBot* activates by posting from user except own. This feature are based on the Hubot `hear` method. When some user posts a message in the β_i, *CiBot* constructs message e, and send it to the relay server B_r.

In the posting message, *CiBot* waits the message from the relay server B_r. For this, *CiBot* listens HTTP requests as a REST service. The endpoint of the REST service on *CiBot* is `/api/cibot/messages`. This endpoint only accepts HTTP `POST` method for bridging messages. When the endpoint accept a HTTP request, *CiBot* parses the request body message e', and posts the message o' to the specified room with updating *CiBot*'s icon. The icon of *CiBot* are updated by e' from the relay server B_r.

Finally, to bridge chat services conceal various information from the users, e.g., joined members, connected rooms, and so on. Ordinary, the user can see the joined members from the application view of the chat services. Therefore, *CiBot* has the interface to answer such questions. In the implementation, *CiBot* provides the interface to show the joined members and connected rooms.

4 Case Studies

This section shows the case study of our proposed system *CiBridge*. We assume that C_x is Slack, and C_y is Rocket.Chat. Also, Table 2 shows the users in both chat services and each room name. Note that, the users `alice` are the different users with the same username. Then, the both `alice` install suitable *CiBot*s to $r_{x\to y}$ and $r_{y\to x}$, respectively. The section shows the messages on each chat services following the below scenarios.

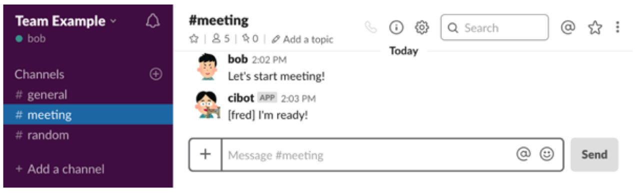
Table 2. The users in each chat service

	Chat service	Room name	User names			
C_x	Slack	Meeting	`alice`	`bob`	`charlie`	`dyran`
C_y	Rocket.Chat	Briefing	`alice`	`eric`	`fred`	`greg`

1. `bob` posts the message in the $r_{x\to y}$,
2. `alice` in C_y posts the message in the $r_{y\to x}$, and
3. `eric` mentions to `alice` of C_x in $r_{y\to x}$.

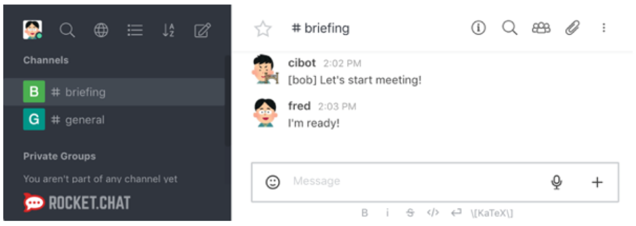
4.1 Case Study 1: A Bridging a Message Example

This case study is a simple example of bridging the message between two chat services. This case study shows the bridged messages in the practical environment. Figure 5 presents the screenshots of chat applications, Bob's view, and Fred's view. In Fig. 5(a), Bob post a message "Let's start meeting!". Then, Fred receives the message via *CiBridge*, shown in Fig. 5(b). Also, *CiBridge* bridges the response message "I'm ready!" from Fred in both views.



bob 2:02 PM
Let's start meeting!
cibot APP 2:03 PM
[fred] I'm ready!

(a) The chat view of Bob



cibot 2:02 PM
[bob] Let's start meeting!
fred 2:03 PM
I'm ready!

(b) The chat view of Fred

Fig. 5. The chat views of Bob and Fred

4.2 Case Study 2: The Same Name User in the Bridged Chat Services

This case study shows how to distinguish the same name users in the bridged chat services. From Table 2, `alice` is in C_x and C_y, and two `alice` are the different users with the same username. The same usernames are usually not accepted, however, in the environment of bridging two chat services, same usernames may exist. Therefore, we should identify the two same name users by their icons. Figure 6 shows the chat application views of Alice. We can identify two alice by their icons, and we can see that two alices have a conversation through *CiBridge*.

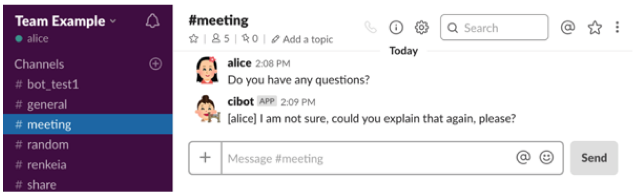
4.3 Case Study 3: Mention to the Name Overlapped User Under the Bridging Services

This case study presents how to mention to the overlapped usernames beyond the chat services. In $r_{y \to x}$ of C_y, `eric` mentions to `alice` of C_x. For this, `eric` post a message "`@cibot_alice Please review #32`", to request a review the issue #32 to `alice` in C_x, shown in Fig. 7. `@cibot_alice` is the keyword of the mention to `alice` in bridged chat services (Fig. 7(b)). *CiBridge* converts the keyword to the general form and post the message, shown as Fig. 7(a).

5 Related Works

After all, the message bridging is just the action by trigger. There are two famous trigger action frameworks, IFTTT[10] and Zapier[11]. Both framework are support

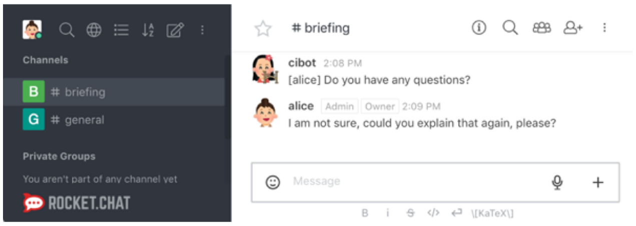
[10] https://ifttt.com/.
[11] https://zapier.com/.



alice 2:08 PM
Do you have any questions?

cibot APP 2:09 PM
[alice] I am not sure, could you explain that again, please?

(a) The chat view of Alice in C_x

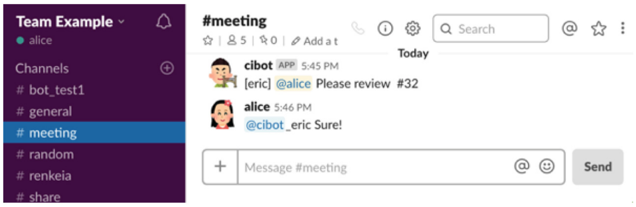


cibot 2:08 PM
[alice] Do you have any questions?

alice Admin Owner 2:09 PM
I am not sure, could you explain that again, please?

(b) The chat view of Alice in C_y

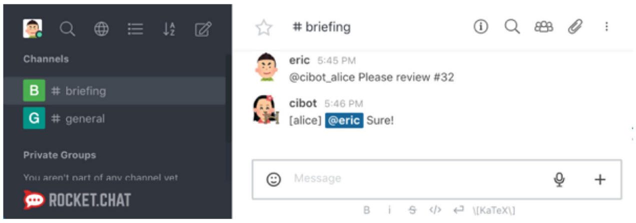
Fig. 6. The chat views of both Alice



cibot APP 5:45 PM
[eric] @alice Please review #32

alice 5:46 PM
@cibot_eric Sure!

(a) The chat view of Alice in C_x



eric 5:45 PM
@cibot_alice Please review #32

cibot 5:46 PM
[alice] @eric Sure!

(b) The chat view of Eric

Fig. 7. The chat views of Alice and Eric

quite many chat services as a trigger. We can solve the problems of the paper using them. However, the trigger action framework cannot support the situation such as overlapped usernames in the chat services, shown in the case study 2 and 3. Also, the users cannot identify the bridged users with their icons.

6 Conclusion

There are many chat services in the world such as Slack, Skype, gitter.im, etc. Using the multiple services needs trivial, however, troublesome works, such as switching services. For solving the problem, there is the exchanging service among several chat services, such as sameroom.io. However, those services do not support to identify the users by their icons.

We built a bridging service *CiBridge* between two chat services to hold the original user icons under the bridging environment. For the proposed method, the user can identify the users by not only their name but also the icons.

In our future works, we will conduct the experimental evaluation in the practical environment.

References

1. Mcdougall, S.J., Curry, M.B., de Bruijn, O.: Behavior research methods. Instrum. Comput. **31**(3), 487 (1999)
2. Ng, A.W., Chan, A.H.: Ind. Eng. Res. **4**(1), 1 (2007)
3. Ng, A.W., Chan, A.H.: Proceedings of Interenational Multiconference Engineers and Computer Scientists vol. 2, pp. 19–21 (2008)
4. Fielding, R.T.: Architectural styles and the design of network-based software architectures. Ph.D. thesis, University of California, Irvine (2000)