Adaptive Creation of Network Applications in the Jack-in-the-Net Architecture

Tomoko Itao[1], Tetsuya Nakamura[1], Masato Matsuo[1], Tatsuya Suda[2]*, and Tomonori Aoyama[3]

[1] NTT Network Innovation Laboratories, Nippon Telegraph and Telephone Corporation (NTT), 3-9-11 Midori-cho, Musashino-shi, Tokyo, 180-8585, Japan
`{tomoko, tetsuya, matsuo}@ma.onlab.ntt.co.jp`
[2] Information and Computer Science, University of California, Irvine, Irvine, CA 92697-3425, USA
`suda@ics.uci.edu`
[3] Information and Communication Engineering, The University of Tokyo, 7-3-1 Hongo Bunkyo-ku, Tokyo, 113-8656, Japan
`aoyama@mlab.t.u-tokyo.ac.jp`

Abstract. The Jack-in-the-Net Architecture (Ja-Net) is a biologically-inspired approach to design adaptive network applications in large-scale networks. In Ja-Net, a network application is dynamically created from a collection of autonomous components called *cyber-entities*. Cyber-entities first establish relationships with other cyber-entities and collectively provide an application through interacting or collaborating with relationship partners. Strength of a relationship is the measure for the usefulness of the partner and adjusted based on the level of satisfaction indicated by a user who received an application. As time progresses, cyber-entities self-organize based on strong relationships and useful applications that users prefer emerge. We implemented Ja-Net platform software and cyber-entities to verify how popular applications (i.e., applications that users prefer) are created in Ja-Net.

1 Introduction

We envision in the future that the Internet spans the entire globe, interconnecting all humans and all man-made devices and objects. When a network scales to this magnitude, it will be virtually impossible to manage a network through a central, coordinating entity. A network must be autonomous and contain built-in mechanisms to support such key features as scalability, adaptability, simplicity,

---

and survivability. We believe that applying concepts and mechanisms from the biological world provides a unique and promising approach to solving key issues that future networks face.

The Jack-in-the-Net Architecture (Ja-Net)[1][2] is a biologically-inspired approach to design adaptive network applications in future networks. The biological concept that we apply in Ja-Net is *emergent behavior* where desirable structure and characteristics emerge from a group of interacting individual entities. In Ja-Net, a network application is dynamically created from a group of interacting autonomous components called *cyber-entities*. A cyber-entity is software with simple behaviors such as migration, replication, reproduction, relationship establishment and death, and implements a set of actions related to a service that the cyber-entity provides. An application is provided through interactions of its cyber-entities. In providing applications, cyber-entities first establish relationships with other cyber-entities and then choose cyber-entities to interact with based on relationships. Strength of a relationship indicates the usefulness of the partner and dynamically adjusted based on the level of satisfaction indicated by a user who received an application. As time progresses, cyber-entities self-organize based on strong relationships resulting in useful emergent applications that users prefer.

In this paper, we describe design and implementation of mechanisms to create applications adaptively in Ja-Net. The rest of the paper is organized in the following manner. Section 2 describes related work. Section 3 describes the overview of Ja-Net Architecture and design of cyber-entities. Section 4 describes experiments on dynamic creation of applications in Ja-Net. Conclusion and future work are discussed in section 5.

2   Related Work

Currently, some frameworks and architectures exist for dynamically creating applications. One such example is Hive [3], where an application is provided through interaction of distributed agents. In Hive, agents choose agents to interact with by specifying the Java interface object that each agent implements. Thus, interaction in Hive is limited to among the agents that mutually implement the interface object of the partner. Unlike Hive, Ja-Net supports ACL (Agent Communication Language) [4] to maximize flexibility in cyber-entity interactions. Bee-gent [5] is another example of a framework to create applications dynamically. It uses a centralized mediator model; a mediator agent maintains a centralized application scenario (logic) and coordinates agent interactions to reduce complexity in multi-agent collaboration. With this centralized mediator approach, Bee-gent restricts the flexibility and scalability of the agent collaboration. Unlike Bee-gent, in Ja-Net, there is no centralized entity to coordinate cyber-entity services, and thus, it scales in the number of cyber-entities. In addition, Ja-Net goes one step further than these architectures by providing built-in mechanisms to support adaptive creation of network applications that reflect user preferences and usage patterns.
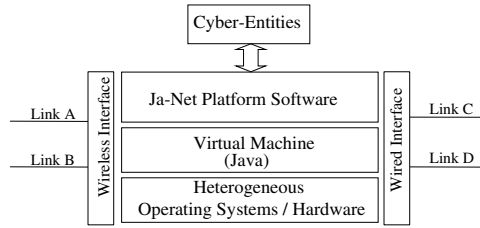
Fig. 1. Ja-Net node structure

Current popular mobile agent systems, including IBM's Aglets [6], General Magic's Odyssey [7], ObjectSpace's Voyager [8] and the University of Stuggart's Mole project [9], adopt the view that a mobile agent is a single unit of computation. They do not employ biological concepts nor take the view that a group of agents may be viewed as a single functioning collective entity.

3   Design of Cyber-Entities

3.1   Overview of the Ja-Net Architecture

Each node in Ja-Net consists of the layers as shown in Figure 1. Ja-Net platform software (referred to as the *platform software* in the rest of the paper) runs using a virtual machine (such as the Java virtual machine) and provides an execution environment and supporting facilities for cyber-entities such as a communication and life-cycle management of cyber-entities. Cyber-entities run atop the platform software. The minimum requirement for a network node to participate in Ja-Net to run the platform software.

A cyber-entity consists of three main parts: *attributes*, *body* and *behaviors*. *Attributes* carry information regarding the cyber-entity (e.g., cyber-entity ID, service type, keywords, age, etc.). The cyber-entity *body* implements a service provided by a cyber-entity. Cyber-entity *behaviors* implement non-service related actions of a cyber-entity such as migration, replication, relationship establishment and death.

3.2   Cyber-Entity Communication

In order to collectively provide an application by a group of cyber-entities, cyber-entities exchange messages during the execution of cyber-entity services. Upon receiving a message, a cyber-entity interprets the message and invokes an appropriate service action and sends the outcome of the action to another cyber-entity that it interacts with. This, in turn, triggers service invocation of those cyber-entities that receive a message. Cyber-entities may also invoke their services based on an event notification. In Ja-Net, various events may be generated triggered by changes in the network or in the real world (such changes may be captured by sensors).

In Ja-Net, to maximize the flexibility in application creation, we adopt Speech Act based FIPA ACL (Agent Communication Language)[4] with extensions specific to the Ja-Net as a communication language of cyber-entities. In the Ja-Net ACL, we define a small number of communicative acts (such as *request*, *agree*, *refuse*, *inform*, *failure*, *query-if*, *advertise*, *recruit*, and *reward*) to facilitate communication between cyber-entities. *Advertise*, *recruit* and *reward* are not in the FIPA ACL communicative acts and specific to the Ja-Net ACL. They are used during the execution of relationship establishment behavior (please see section 3.4 for relationship establishment behavior). In the Ja-Net ACL, an event notification message is also delivered in ACL using *inform* communicative act. Each ACL message exchanged between cyber-entities contains a communicative act and parameters such as *:receiver*, *:sender*, *:in-reply-to*, *:ontology*, *:sequence-id* and *:content*. *:Receiver* and *:sender*, parameters specify the receiver of the current message and the sender of the current message, respectively. *:In-reply-to* specifies to which message it is replying and is to manage the message exchange flow between cyber-entities. *:Ontology* specifies the vocabulary set (dictionary) used to describe the content of the message. *:Sequence-id* specifies a unique identifier of a message sequence in providing an application . A *sequence-id* is generated by a cyber-entity at the initial point of an application and piggy backed by each ACL message exchanged during the application. *:Content* specifies data or information associated with a communicative act in the message. A *:content* parameter is described with Extensible Markup Language (XML)[10].

3.3   Cyber-Entity Body

A cyber-entity service is implemented as a finite state machine. A cyber-entity may have multiple state models and execute them in parallel. Each state model consists of states and state transition rules. A state implements an atomic service action and message exchanges associated with the action (to allow inputting data to and outputting data from a given action in a given state). A state transition rule associated with a state specifies the next state to transit to. When an action in a given state completes, the current state moves to the next state based on the state transition rule.

In sending the outcome of a service action, a cyber-entity may either respond to a cyber-entity that sent the previous message, or send the message to another cyber-entity (or cyber-entities) by selecting a cyber-entity (or cyber-entities) to interact with using relationship (please see section 3.4 for interaction partner selection mechanism). Upon receiving a message from another cyber-entity, a cyber-entity invokes an appropriate state (action) that can handle the message by examining parameters of the incoming message in the following manner. If the parameter *:in-reply-to* is set in the incoming message, it is in response to a previously transmitted message. In this case, the cyber-entity compares the data type of the *:content* in the incoming message with the input data type required by an action (state) where the previous message was transmitted, and invokes the action if it can take the incoming message as its input. If the parameter *:in-reply-to* of the incoming message is null, the incoming message is the first message
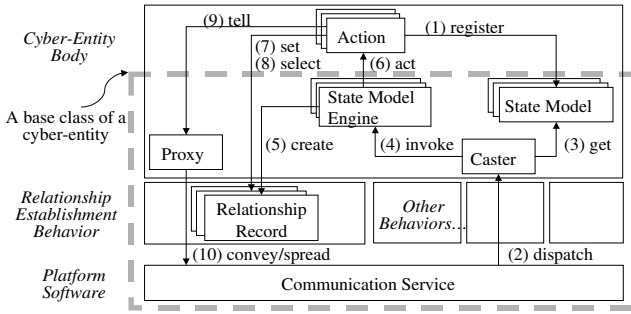
Fig. 2. Function components at a cyber-entity

from the sender cyber-entity. In this case, the receiver cyber-entity examines the current state of a state model that is ready to interact with a new cyber-entity and the initial state of each and every state model that it implements. Among them, a state that can take the incoming message as its input is then invoked.

Figure 2 shows the main function components (classes) of a cyber-entity. In our current design, classes in the cyber-entity *body* except *action* as well as classes in cyber-entity *behaviors* are implemented in a base class of a cyber-entity, and all cyber-entities are derived from the base class. Service actions are implemented by cyber-entity designers and registered with a *state model* (depicted as (1) "register" in Figure 2). *Caster* receives an ACL message from another cyber-entity via the communication service in the platform software (depicted as (2) "dispatch" in Figure 2), examines state models (depicted as (3) "get" in Figure 2) and invokes an appropriate state (action) (depicted as (4) "invoke" and (6) "act" in Figure 2). *State model engine* is a generic class to execute a state model. *Proxy* represents a remote cyber-entity and provides an API to send a message to the remote cyber-entity (depicted as (9) "tell" in Figure 2). The outgoing message is unicast (or multicast)/broadcast by the platform (depicted as (10) "convey/spread" in Figure 2).

3.4 Relationship Management

Relationship Attributes. A relationship may be viewed as (cyber-entity's) information cache regarding other cyber-entities. Table 1 shows example relationship attributes stored in a *relationship record* (depicted as "Relationship Record" in Figure 2) at a cyber-entity. *CE-id* is to uniquely identify a relationship partner cyber-entity. *Action-name* specifies an action of the cyber-entity itself to interact with a relationship partner cyber-entity. *Service-properties* is to store information regarding the service that a relationship partner cyber-entity provides (such as the service type and keywords of a relationship partner cyber-entity). *Access-count* may be incremented when a service message is exchanged with a relationship partner cyber-entity. *Strength* evaluates the usefulness of a

Table 1. Example attributes of a relationship record at a cyber-entity

Attribute	Meaning
CE-id	A globally unique identifier of a relationship partner cyber-entity.
Action-name	An action of the cyber-entity itself that may be used to interact with a relationship partner cyber-entity.
Service-properties	Information regarding the service that a relationship partner cyber-entity provides.
Access-count	The number of interactions with a relationship partner cyber-entity.
Strength	Indication of the usefulness of a relationship partner cyber-entity.

partner cyber-entity and is used to help cyber-entities to select useful interaction partners.

Relationship Establishment. Cyber-entities first establish relationships with other cyber-entities to interact with. For instance, a cyber-entity that has just migrated to a new node may broadcast an *advertise* message specifying information regarding the sender cyber-entity (e.g., service type and/or attributes) to establish relationships with nearby cyber-entities. Upon receiving an *advertise* message, a cyber-entity creates a new *relationship record* (depicted as (5) "create" in Figure 2) and stores the sender cyber-entity's CE-id and information obtained from the incoming *advertise* message in the relationship record. Additional information about a relationship partner cyber-entity obtained through interaction may be stored in the *Service-properties* of its relationship record (depicted as (7) "set" in Figure 2). Alternatively, a cyber-entity may broadcast a *recruit* message specifying conditions on a partner (e.g., service type and/or attributes required for a partner cyber-entity). A cyber-entity that receives a *recruit* message responds with an *inform* message containing its own information if it satisfies conditions specified in the *recruit* message. Through this interaction, the sender cyber-entity and the receiver cyber-entity of the *recruit* message may mutually establish a relationship with each other.

Partner Selection. In selecting an interaction partner cyber-entity (or cyber-entities), a cyber-entity may specify one or more relationship attributes as keys and retrieve its relationship records that match the specified keys (depicted as (8) "select" in Figure 2). If there are multiple relationship records that match the keys, a cyber-entity narrow these relationship records based on relationship strengths so that the cyber-entity interacts more often with cyber-entities with stronger relationships. If there is none or less relationship record that matches the keys, a cyber-entity attempts to discover new cyber-entities by broadcasting an *advertise* message or a *recruit* message to nearby cyber-entities.

Strength Adjustment. In Ja-net, a user indicates in *happiness* the degree of his/her satisfaction with the received application. When a user receives an application, the user creates a *reward* message and sets *happiness* value in the message content. The *reward* is back propagated along the message exchange sequence from a cyber-entity at the end point of the application to a cyber-entity at the initial point of the application. In order to remember a back propagation

path, each cyber-entity records the previous and the next cyber-entities in the message sequence along with the corresponding *sequence-id* (which is obtained from ACL *:sequence-id* parameter). Upon receiving a *happiness* value in the *reward* message, each cyber-entity modifies the strength of relationships regarding cyber-entities that it interacted with in providing an application. If a user likes the application, positive *happiness* value is returned and the strength value is increased. If a user dislikes the application, negative *happiness* value is returned and the strength value is decreased. If user is neutral or no *happiness* value is returned, there is no change in the strength value. Therefore, cyber-entities that collectively provide a popular application (i.e., application that a number of users like) will receive a positive *happiness* value more often and strengthen the relationship among themselves, while relationships among cyber-entities that provide a not-so-popular application are weakened.

Group Formation. In order to allow users to explicitly request for an application, cyber-entities collectively providing an application form a group when the relationship strengths among themselves exceed a predetermined threshold value. Once a group is formed, a unique group ID, as well as human-readable application name, is assigned to each group member cyber-entity. Thus, users can request for a group service either by a unique group ID or by a human-readable application (group) name.

4   Experiments on Dynamic Application Creation

In order to verify dynamic creation of applications in Ja-Net, we implemented multiple cyber-entities, as well as platform software, based on the design described in section 3 and performed basic experiments. In our experiments, various realistic scenarios were simulated through running multiple cyber-entities and multiple platform software on computers. Our implementation and experiments are explained below.

4.1   Application Implementation

In applications that we implemented, we consider popular public spots such as the New York City's Times Square, a theater in the nearby Broadway theater district and a cafe on the New York City's Fifth Avenue. A number of people (users) visit these locations, stay there for a while (doing, for instance, window shopping, watching a show, having some coffee at a cafe), and leave. Assume that these users carry a mobile phone or a PDA that is capable of running cyber-entities and communicating with other mobile phones and PDAs in an ad-hoc manner. Assume also that shops in these area implement cyber-entities related to their service and run these cyber-entities on a computer in the shop. In addition, some users may implement their own cyber-entities or have down loaded and carry cyber-entities in their mobile phones and PDAs from where they visited earlier in the day. Various cyber-entities join/leave to/from each location
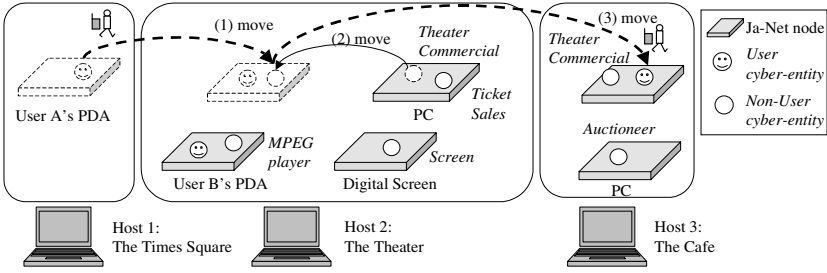
Fig. 3. Overview of the Ja-Net experiment system

according to the movement of users, which triggers actions and interactions of other cyber-entities.

Figure 3 shows the overview of the Ja-Net experiment system. Each host represents different public spot, such as the Times Square, a theater and a cafe, respectively, and each Ja-Net node represents a PC, a device or user's PDA that is supposed to be present at a location that its host computer represents. User's PDA runs a cyber-entity (user cyber-entity) representing the user. Each node runs one or more cyber-entities as described below. User A's PDA at the Times Square runs a user cyber-entity representing user A. A PC at the theater runs a *TheaterCommercial* cyber-entity that stores information of a theater show (assume that this information contains a URL of a commercial video clip of the theater show and a button to request for a ticket purchase as well as other show information) and a *TicketSales* cyber-entity that issues a theater show ticket and generates a certification of ticket purchase. A digital screen in the theater runs a *Screen* cyber-entity that displays image or video on the digital screen. A PDA of another user B at the theater runs a user cyber-entity representing user B and a *MPEGplayer* cyber-entity (assume that it is down loaded by user B earlier in the day). A PC in the cafe runs an *Auctioneer* cyber-entity that purchases commercial products from other cyber-entities and sells them at auction.

In order to capture users' behaviors in our experiments, we defined two types of events. NODE_ARRIVAL is an event generated by platform software when a Ja-Net node arrives at a new location. USER_BROWSING is an event that is generated by a user cyber-entity when a human user shows interests in the information displayed on his/her PDA. (For instance, this event is generated when a user scrolls the window up and down on his/her PDA). These events are broadcast to cyber-entities in the same location (i.e., in the same host).

Example Application Sequence. Figure 4 shows an message sequence of an application we implemented (referred to as *ticket sales* sequence). In this sequence, a *TheaterCommercial* cyber-entity displays information of a theater show (depicted as (1) "inform" in Figure 4) on user's PDA. Suppose that the user is interested in the show and sends a request for ticket purchase to the *TheaterCommercial* cyber-entity (depicted as (2) "request" in Figure 4). Since the *TheaterCommercial* cyber-entity only stores information of the show and does
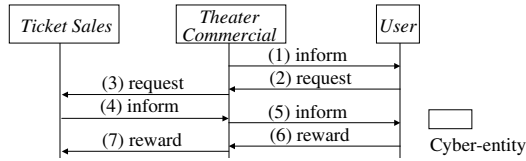
Fig. 4. An example of an application sequence (*ticket sales*)

not implement a ticket sales service, it forwards the request to a *TicketSales* cyber-entity that it has relationship with (depicted as (3) "request" in Figure 4). Upon receiving a forwarded request for a ticket purchase, the *TicketSales* cyber-entity issues a certificate for ticket purchase and sends the certification to the *User* cyber-entity via the *TheaterCommercial* cyber-entity (depicted as (4) "inform" and (5) "inform" in Figure 4 respectively). When a human user obtains a ticket (i.e., a certificate of a ticket purchase) from the *User* cyber-entity, he/she expresses the level of satisfaction as the *happiness* value. *User* cyber-entity then creates a *reward* message and sends it to the *TheaterCommercial* cyber-entity, which, in turn, forwards the *reward* message to the *TicketSales* cyber-entity (depicted as (6) "reward" and (7) "reward" in Figure 4). The relationship strength between the *TheaterCommercial* cyber-entity and the *TicketSales* cyber-entity is adjusted based on the *happiness* value.

4.2   Experimental Results

In our experiments, we only implemented the *body* (i.e., services) and *relationship establishment behavior* of cyber-entities. Thus, cyber-entities were manually moved to simulate their migration behavior when necessary in our experiments. When an experiment starts, cyber-entities initially do not have relationship with any other cyber-entities. Each cyber-entity dynamically establishes relationships with cyber-entities in the same location (i.e., in the same host) by broadcasting an *advertise* message or a *recruit* message. Once relationships are established, cyber-entities start interacting with relationship partners and collectively provide applications. We performed several experiments to examine dynamic application creation in Ja-Net. Our experiments are described below.

Experiment 1. In this experiment, we manually moved a node representing user A's PDA and a user cyber-entity representing user A (on user A's PDA) from the Times Square to the theater (depicted as (1) "move" in Figure3) to simulate user A's movement and observed that an application emerged through interactions of cyber-entities (detailed explanation is described below). Upon arriving at the theater, user A's PDA generated NODE_ARRIVAL event and broadcast the event to all cyber-entities in the theater. Upon receiving the event, a user cyber-entity on user A's PDA, one of the cyber-entities in the theater, broadcast an *advertise* message to cyber-entities in the theater. Then, upon receiving the *advertise* message, the *TheaterCommercial* cyber-entity (on PC) established a relationship with a user cyber-entity (on user A's PDA) and sent theater show

Fig. 5. A screen snap shot of application windows

information that it stores to the user cyber-entity (on user A's PDA), which in turn displayed the theater show information on user A's PDA. At this moment, user A scrolled a window on his/her PDA. (In our experiments, we, human operators conducting the experiment, scrolled a window up and down on user A's PDA). This generated a USER_BROWSING event. The event was broadcast to cyber-entities in the theater. In this experiment, we assumed that the *Theater-Commercial* cyber-entity had a relationship with a *MPEGplayer* cyber-entity on a PDA of another user B. Thus, the *TheaterCommercial* cyber-entity, upon receiving the USER_BROWSING event, sent theater show information that it stores to the *MPEGpalyer* cyber-entity. The *MPEGplayer* cyber-entity invoked its service and accessed a commercial video clip of a theater show using a URL included in the theater show information. In this experiment, we also assumed that the *MPEGplayer* cyber-entity had a relationship with a *Screen* cyber-entity on the digital screen. Thus, the *MPEGplayer* cyber-entity sent the outcome of its action to the *Screen* cyber-entity. Consequently, the *Screen* cyber-entity displayed the commercial video clip of a show on the digital screen in the theater.

Figure 5 shows application windows displayed by each node. A window of user A's PDA (on the left) displays theater show information and a window of the digital screen (in the center) displays a commercial video clip of a show.

Experiment 2. In this experiment, while theater show information is displayed on user A's PDA, user A clicked a ticket purchase button. (In our experiments, we, human operators conducting the experiment, clicked the button on user A's PDA). A user cyber-entity on user A's PDA generated a *request* message for a ticket purchase and sent it to the *TheaterCommercial* cyber-entity (on PC). Then, we observed interaction between the *TheaterCommercial* cyber-entity and the *TicketSales* cyber-entity (on PC) shown in Figure 4. User A then

received a certificate of ticket purchase. At this point, we have demonstrated that an application was created upon receiving a *request* message from a user.

Next, in order to show that different applications emerge in different environments (i.e., environments where different sets of cyber-entities exist), we simulated the *TheaterCommercial* cyber-entity migrated to user A's PDA (i.e., *TheaterCommercial* cyber-entity was manually moved to user A's PDA, which is depicted as (2) "move" in Figure 3), and also simulated user A's movement from the theater to the cafe (depicted as (3) "move" in Figure 3). Upon arriving at the cafe, user A's PDA generated a NODE_ARRIVAL event and broadcast the event to cyber-entities in the cafe (including the *TheaterCommercial* cyber-entity on user A's PDA). Upon receiving the event, the *TheaterCommercial* cyber-entity broadcast an *advertise* message to cyber-entities in the cafe. Upon receiving the *advertise* message, the *Auctioneer* cyber-entity (on PC in the cafe) established a relationship with the *TheaterCommercial* cyber-entity and invoked its service action to purchase a commercial product (i.e., a show ticket in this case). Then, a *request* for a ticket purchase is sent from the *Auctioneer* cyber-entity to the *TheaterCommercial* cyber-entity, which in turn forwarded the *request* message to the *TicketSales* cyber-entity (on PC in the theater) following the same sequence shown in Figure 4 except the *Auctioneer* cyber-entity played the role of "User" in this case. The *Auctioneer* cyber-entity received a certificate of ticket purchase and it provided auction service to users by selling the ticket. This experiment verified that the same cyber-entity may provide different applications by interacting with different cyber-entities.

Experiment 3. In order to examine the group formation mechanism proposed in this paper, we artificially created a large number of requests on user A (in the cafe) to purchase a show ticket and sent them to the *TheaterCommercial* cyber-entity (on user A's PDA in the cafe). We assumed in this experiment that user A is satisfied with a ticket purchased from the *TheaterCommercial* cyber-entity, and thus, user A always returned a positive *happiness* value. As time progresses, we observed that the relationship strength from the *TheaterCommercial* cyber-entity to the *TicketSales* cyber-entity (on PC in the theater) as well as the relationship strength from the *TicketSales* cyber-entity to the *TheaterCommercial* cyber-entity gradually increased. When both relationship strengths exceeded a predetermined threshold value, a group of the *TheaterCommercial* cyber-entity and the *TicketSales* cyber-entity was formed. Once a group is formed, the *TheaterCommercial* cyber-entity, an initial point of the application, sent an *advertise* message containing the group ID, and user A was able to invoke the group service by sending a *request* message containing the group ID to the *TheaterCommercial* cyber-entity.

Through experiments 1–3, we verified that through the mechanisms we proposed in this paper, Ja-Net dynamically creates applications that reflect user preferences and usage patterns. Several applications emerged in our experiments, and only popular applications (i.e., applications that users prefer) formed a group.

5   Conclusion and Future Work

The Jack-in-the-Net (Ja-Net) Architecture is a biologically-inspired approach to design and implement adaptive network applications. Ja-Net is inspired by and based on the Bio-Networking Architecture project in University of California, Irvine [11][12]. This paper described design of cyber-entities and key mechanisms used in Ja-Net for cyber-entity interaction and relationship management. This paper also examined and verified these key mechanisms through experiments.

As for future work, we plan to support interaction protocols between cyber-entities to allow more complex collaboration. We also plan to investigate various algorithms for relationship strength adjustment and partner selection in addition to these described in this paper. Various algorithms will be empirically evaluated for their efficiency in creation and provision of adaptive applications. Experimental study through implementation and deployment of a large scale applications will also be conducted.

References

1. T. Suda, T. Itao, T. Nakamura and M. Matsuo, "A Network for Service Evolution and Emergence," Journal of IEICEJ, Invited Paper, Vol.J84-B, No.3, 2001.
2. T. Itao, T. Nakamura and M. Matsuo, T. Suda, and T. Aoyama, "Service Emergence based on Relationship among Self-Organizing Entities," Proc. of the IEEE SAINT2002 (Best Paper), Jan., 2002.
3. N. Minar, M. Gray, O. Roup, R. Krikorian, and P. Maes, "Hive: Distributed Agents for Networking Things," Proc. of the ASA/MA '99, Aug., 1999.
4. Foundation for Intelligent Physical Agents, "FIPA Communicative Act Library Specification, 2000," available at http://www.fipa.org/
5. T. Kawamura, Y. Tahara, T. Hasegawa, A. Ohsuga and S. Honiden, "Bee-gent: Bonding and Encapsulation Enhancement Agent Framework for Development of Distributed Systems," Journal of the IEICEJ, D-I, Vol. J82-D-I, No.9, 1999.
6. D. B. Lange and M. Oshima, "Programming & Deploying Mobile Agents with Java Aglets," Addison-Wesley, 1998.
7. Odyssey Home Page. http://www.genmagic.com/technology/odyssey.html
8. Voyager Home Page. http://www.objectspace.com/products/voyager/
9. Mole Project Home Page,
   http://inf.informatik.uni-stuttgart.de/ipvr/vs/projekte/mole.html
10. XML web site, http://www.xml.org
11. The BNA Project Home Page. http://netresearch.ics.uci.edu/bionet
12. Michael Wang and Tetsuya Suda, "The Bio-Networking Architecture: A Biologically Inspired Approach to the Design of Scalable, Adaptive, and Survivable/Available Network Applications," Proc. of the IEEE SAINT2001, Jan., 2001.