Studying Complex Interactions in Real Time: an XMPP-based Framework for Behavioral Experiments

Dan Mønster[1,2,3]
[1]*Interacting Minds Centre, Aarhus University, DK-8000 Aarhus C, Denmark*
[2]*Cognition and Behavior Lab, Aarhus University, DK-8210 Aarhus V, Denmark*
[3]*Department of Economics and Business Economics, Aarhus University, DK-8210 Aarhus V, Denmark*
danm@econ.au.dk

Keywords: Real-Time, Behavioral Experiment, XMPP, Python, Social Network, Complexity.

Abstract: The study of human behavior must take into account the social context, and real-time, networked experiments with multiple participants is one increasingly popular way to achieve this. In this paper a framework based on Python and XMPP is presented that aims to make it easy to develop such behavioral experiments. An illustrative example of how the framework can be used is also presented. This example is a real experiment, which is currently gathering data in the lab.

1 INTRODUCTION

Humans are social beings that are connected to each other in ties of kinship, friendship, and other affiliations in groups, organizations and societies at different scales, ranging from families and friendship groups to workplaces, cities, nation states and global communities. These groups are mediated by both physical and symbolic interactions in real and virtual environments, and are often represented as social networks.

Using social networks to describe interactions between humans is a long tradition in sociology (Granovetter, 1973), and in recent years it has also been used in cognitive science (Baronchelli et al., 2013) to study, e.g., the emergence of social linguistic conventions (Centola and Baronchelli, 2015). Social network structure may influence many other areas of our lives, for example, recent simulation studies indicate that the structure of the social network and the kind of social learning that takes place play a combined role in determining team performance (Barkoczi and Galesic, 2016).

Interconnected networks are relevant not only for the study of complex interactions between humans, but also in complex coordinated activities between humans and machines, e.g., when humans interact with cloud services or IoT devices (Dustdar et al., 2016). It is therefore important to be able to study these interactions both in the field and in the lab. In economics there is a long tradition of using lab experiments based on game theory to study human behavior and decision making. Several tools exist for running such experiments (see section 1.1), but few of them enable researchers to connect participants in experiments in arbitrary network structures and to dynamically change these structures.

Recent studies of interacting groups suggest that part of the key to understanding successful collaboration is connected to the dynamics of how group members interact in real time (Mønster et al., 2016; Wallot et al., 2016; Fusaroli and Tylén, 2016). Another recent study indicates that real-time interaction influences the efficiency and fairness of collective outcomes (Hawkins and Goldstone, 2016).

To facilitate and expand these kinds of studies there is a need for an easy way for researchers to design, build and deploy real-time networked experiments both on the web and in the lab. This paper presents a solution to this challenge for lab-based experiments, and although the solution can also be extended to the web, other existing solutions are better suited for this purpose. There are differences between lab-based and web-based experiments, and both approaches have distinct advantages and challenges (Woods et al., 2015).

The solution described in the present paper aims to satisfy the need for an easy way to develop software for lab experiments where participants interact in real time. The approach is to provide a minimal framework that can be integrated with researchers own code on the client side (e.g., existing GUI or peripheral equipment)

Presented at COMPLEXIS 2017, www.complexis.org.

and on the server side (computations, game logic) as well as allowing for computer-based agents to interact with human participants.

1.1 Related works

PsychoPy (Peirce, 2007; Peirce, 2009) is a Python based open source software tool to build experiments with frame accurate timing for psychophysics type experiments. PsychoPy is cross platform and runs on Windows, Mac OS and Linux. It uses OpenGL for fast 2D graphics, supports multiple monitors, monitor calibration and includes many standard visual and auditory stimuli. PsychoPy also supports many external devices such as response boxes, parallel and serial ports, eye trackers, and more.

PsychoJS is a recent initiative to bring PsychoPy functionality to the web browser using JavaScript and pixi.js and WebGL for high-performance 2D rendering in the browser.

One of the most used tools for this type of experiments has been the Zürich Toolkit for Readymade Economic Experiments: z-Tree (Fischbacher, 2007). z-Tree runs on Windows computers and employs a client-server architecture, with a client application called z-Leaf and a server application called z-Tree. The programming model of z-Tree is somewhat idiosyncratic, using a spreadsheet like construct called tables which holds most variables and where scoping rules and the scoping operator controls referencing variables from other tables.

A recent alternative is oTree (Chen et al., 2016) which is an open source Python based solution, which like z-Tree is a client-server architecture, that uses a web browser as the client through Django. This has the advantage that experiments can run on most platforms in the lab, online and in the field.

Experimental Tribe, or XTribe (Caminiti et al., 2013) is a web platform for programming and advertising so-called games with a purpose or citizen science games, through the Xtribe web site. Xtribe functionality is similar to oTree, but is tied to the Xtribe platform, which also serves to recruit participants. Game backends (called game managers) in Xtribe can be programmed in PHP, JavaScript (node.js), Python or JSP.

The most recent addition to the family of browser-based multi-participant experiment platforms is nodeGame (Balietti, 2016), which uses JavaScript for both client and server. nodeGame has many of the same features as oTree and XTribe and is built on node.js.

A system for real-time multiplayer experiments on the web including a physics engine (Hawkins, 2015) has also been developed. The system, called MWERT, is also based on node.js, and uses the HTML 5 canvas to display client-side graphics. It also includes a physics engine, so that players position in a virtual world can be used in experiments.

The solution described in the present paper gives researchers a way to develop real-time multi participant experiments using a widely used programming language (Python) in a flexible manner. In this regard it is similar to oTree, nodeGame, Xtribe and MWERT, although unlike these solutions it is *not* web based. This is a deliberate design, although when development of the present solution was started the author had no knowledge of these other solutions, most of which have only recently been published. It seems that most of these related works were developed in parallel with little or no knowledge of the other solutions.

2 SOLUTION

This paper describes a solution that was needed for several novel experiment designs at Cognition and Behavior Lab at Aarhus University in late 2014. At the time, no existing solution was found to satisfy all the requirements needed for these designs, which included the ability to embed players in different (social) network topologies, real-time interaction between players, interactive graphics and use of audio, use of server-side calculations, and the ability to mix human participants and software agents.

Since multiple projects had similar requirements, and in the interest of being able re-use the solution for future projects, a general solution was sought. Cognition and Behavior Lab is an interdisciplinary lab used by researchers from several disciplines at Aarhus University, including, but not limited to psychology, economics, cognitive science, marketing, political science, neuroscience, and linguistics. As a consequence many different platforms for developing behavioral experiments are used, but open source platforms have become increasingly popular – chiefly among these is PsychoPy (Peirce, 2007). Part of the infrastructure at Cognition and Behavior Lab is the Computer-based Interaction Lab, with 33 computers. These computers have PsychoPy installed, and many of the researchers using the lab and programming their own experiments are well versed in Python. A wide variety of experiments have used PsychoPy[1] in a range of different disciplines. For this reason it was decided to implement the initial solution in Python. With a local user-base

[1] At the time of writing the two main PsychoPy publications (Peirce, 2007; Peirce, 2009) had 1130 citations, according to Google Scholar.

of lab researchers and research assistants who could already develop for PsychoPy, the crucial missing ingredient was an easy-to-use networking component that would allow communication between several computers, each running an application that a human participant can interact with. Such an application will be denoted a *client application* or simply a client.

2.1 Design objectives

Based on the specific projects and with an aim to support experiments involving interaction between participants in general, the following design objectives were developed:

Simple interface. The user should not be required to understand object oriented programming, or deal with concurrency and multi-threading issues.

High performance. Since one of the original objectives was to enable game-like real-time interactions, low latency is needed. A latency below 100 ms is generally needed in first-person-shooter games (Claypool and Claypool, 2006) and for VoIP (Markopoulou et al., 2002), so delays should be kept below this level. For larger experiments involving hundreds of participants, scalability and high throughput is also required.

Network protocol agnostic. The solution should be able to work in a wide range of network environments, i.e., in other labs, over the general Internet, and behind firewalls and NAT devices. Hence it is desirable to be able to use, e.g., both TCP and HTTP as transport protocols.

Open. Since researchers use a range of different tools and technologies, an open standard, preferably with open source implementations, is preferred over closed and commercial alternatives.

After evaluating several alternative technologies, it was decided to develop a solution based on the The eXtensible Messaging and Presence Protocol (XMPP) (Saint-Andre, 2011a; Saint-Andre, 2011b). XMPP was originally designed for instant messaging (IM), but has been used as a core enabling technology for multi-player games (Lee, 2004), Internet of Things (IoT) (Schuster et al., 2014), as well as cloud computing (Bernstein et al., 2009) and grid computing (Albano et al., 2015) (see (Hornsby and Walsh, 2010) for an overview). It has thus been demonstrated that the XMPP protocol can be used in a wide variety of applications where the common denominator is connecting humans and/or systems, often with real-time constraints and on a massive scale. Furthermore, the XMPP protocol is open, extensible and with a wide variety of server, client and library implementations[2], many of which are open source. XMPP libraries exist for every major programming language, and for Python there are around 10 different libraries to choose from.

2.2 The XMPP protocol

XMPP is a protocol for exchanging messages over a network in eXtensible Markup Language (XML) between two entities in a client-server architecture. Entities are XMPP clients or the XMPP server to which the clients are connected. Only the simple example of one server with several clients will be considered here, but it is noted that XMPP is a decentralized protocol and it is possible to set up server-to-server connections and gateways to other protocols.

Entities in XMPP are identified by a unique identifier, and since XMPP is the standardization of the original Jabber protocol, this identifier is referred to as the Jabber ID (JID). The JID is modeled after the syntax used for Internet e-mail addresses, and have the general form `user@domain/resource`, where the resource identifier is optional, but may, e.g., be used to identify different devices held by the same user. XMPP exchanges XML elements, called stanzas, between entities, and there are three stanzas defined in the XMPP core protocol (Saint-Andre, 2011a), viz. the `<presence/>` stanza used to indicate availability of the entity, the `<message/>` stanza used to send a message from one entity to another, and the `<iq/>` stanza used to send information and queries between entities.

An XMPP session is initiated by the client and a `<stream/>` element is negotiated and set up between the client and the server. This `<stream/>` is the root to which all of the XML stanzas in the stream belong. When the session is established and the client is authenticated further stanzas can be exchanged. When XMPP is used for IM, the client retrieves the *roster* (contact list) associated with the JID of the client, using an `<iq/>` stanza, and then sends a `<presence/>` stanza to indicate that the user is online and available. Communication between two clients can now commence, and is achieved by sending and receiving `<message/>` stanzas over the established XML stream.

2.2.1 XMPP server

The XMPP server is a key element in setting up the solution, since the server is responsible for client authentication and routing of messages. Since all messages pass through the server, it is also a crucial element in

[2] See, e.g., https://xmpp.org/software for a list.

ensuring low latency and scalability — two of the key objectives. For distributed experiments, where some clients may be behind NAT devices and firewalls, end-to-end communication can still be guaranteed as long as the clients can connect to the server and establish an XMPP stream.

Many different XMPP server implementations exist that can be deployed locally or as hosted solutions by commercial providers. In addition there are many public XMPP servers, where anyone can set up accounts. For the solution described in this paper a locally deployed XMPP server was the best choice, since it guarantees lower network propagation latency than a remotely hosted solution, and provides a more stable environment for testing and experimenting under controlled conditions.

An ejabberd Community Server[3] was installed on a virtual server running Ubuntu Linux hosted by the IT Department at Aarhus University. The command line control interface to ejabberd provided an easy way to create identities in the form of JIDs through a script. In this way it was easy to create an arbitrary number of unique JIDs and passwords that are private to a particular research project.

2.2.2 XMPP library

There are several XMPP libraries implemented in Python or with Python bindings, which could be used for the solution. An early version of the solution was implemented using xmpp.py (Nezhdanov and Rasmussen, 2013), but this library was later replaced by SleekXMPP (Fritz and Stout, 2016) which proved to be more stable, better documented and with a larger community.

2.3 Simple interface to XMPP

Most researchers (in fields other than computer science) who develop their own software are self-taught. A recent study among scientists showed that over 95% are self-taught (Hannay et al., 2009), and Greg Wilson, who has taught software development to scientists over the past two decades, came to the conclusion that more 'advanced' topics such as object-oriented programming, XML, and Make had to be dropped from the curriculum (Wilson, 2016). Hence, for the solution to be adopted by the intended user base, it must present researchers with a simple interface to XMPP, that does not require them to understand the intricacies of the XMPP protocol or XML, and does not require them to deal with issues of multithreading and asynchronous calls.

[3] https://www.ejabberd.im/

Consequently, a simple Python class called NetworkingClient was developed to encapsulate all XMPP details except the client's JID and password, which are passed to the `__init__()` method upon instantiation of an object of the NetworkingClient class. The following code shows how the class can be used in Python code to create a new variable that holds an instance of the class:

```
from NetworkingClient import NetworkingClient

jid = 'client_001_exp_008@my.local.server'
secret = 'V1nxtEcixj1TcYRS'
network = NetworkingClient(jid, secret)
```

A client only needs to know its own JID and password. The XMPP server address is encoded in the JID (in the example: `my.local.server`). The NetworkingClient class also contains methods for sending messages to other XMPP entities identified by a JID, as in this example:

```
receiver = 'client_002_exp_008@my.local.server'
network.send_message(to = receiver,
                     sender = jid,
                     message='Hello!',
                     subject='')
```

In this example the message subject has been left empty, but it can conveniently be used to indicate the kind of message sent (see section 3). The ability to connect to and authenticate with an XMPP server, and to send messages provide sufficient capabilities to implement all the experimental designs for which the solution was primarily developed. The NetworkingClient class also contains methods to manage presence and roster, but these additional methods will not be discussed here.

2.3.1 Message handling

The core functionality needed for the solution is the ability to send and receive messages between any two entities, and if two clients know each others' JIDs, then this functionality is accomplished. Knowledge of other clients' JIDs can be obtained using XMPP's presence and roster functionalities, or JIDs can be known in advance, or controlled by a special client. In the example presented in section 3 the latter method is used, and this special client is called the *experiment server* (not to be confused with the XMPP server).

The NetworkingClient class has a listener running in a separate thread and uses a queue to store messages in the order they are received. The calling code can poll to check for messages and then invoke a function to handle the messages:

```
if network.check_for_messages():
    msg = network.pop_message()
    message_handler(msg)
```

This design means that the calling code does not need to take concurrency issues into account.

3 RESULTS

The NetworkingClient class has been used to implement several experiments where real-time interaction between participants was needed. In the following, one of these, called *networked problem solving* will be described in detail. The purpose of the 'networking problem solving' experiment is to study how the network topology influences how people solve a joint problem, and how solutions to the problem propagate in the network. At the time of writing, data is still being gathered in the experiment, and the results of the experiment will be published elsewhere. Here the focus will be on how the solution described in the present paper was used to facilitate the experiment. The problem to be solved by the participants in the experiment was framed as a collaborative game in which pairs of participants together had to create a four-tone melody on a shared keyboard with five tones (see Figure 1). The terms player and participant will be used interchangeably to refer to the human participants, while the term game client refers to the software that the players use.

3.1 Experimental session

The game is played by an even number of players, each using a game client, and the gameplay is controlled by an experiment server (or game server). Both the game client and game server use instances of the developed NetworkingClient to send and receive XMPP messages.

In an experimental session the game server application is started before the clients, and it uses NetworkingClient to connect to the XMPP server and authenticate, after which it starts listening for messages. When a game client is started it also connects to the XMPP server and authenticates with a matching (JID, password) combination. Since the experiment is conducted in a lab, the JID is found by matching the computer's MAC address to a JID. The game client then sends a register message to the game server:

```
version = 0.5
server_jid = 'server_exp_008@my.local.server'
network.send_message(to=server_jid,
                     sender=jid,
                     subject='register',
                     message=str(version))
```

In the example code the game client's JID and the game server's JID are hardcoded. The XMPP message's subject is used to indicate the type of message that is sent, in this case a 'register' message. The message body contains a string, which for the register message is the game client's version number.

3.2 Message passing

Once the game client has sent a register message to the game server, the server can communicate with the client and all other clients that have registered. The game server maintains a list of registered clients, and when a sufficient number of game clients have registered, it can start the game. In the 'networked problem solving' experiment described here, the game is started by an experimenter who runs and supervises the experimental session, but it could equally well happen automatically.

Before the game is started the experimenter can set a number of parameters in the game server GUI (see Figure 4), most importantly for the description here is the network topology which determines how game clients are connected to each other. The options here are 'fully connected', where any two game clients are connected, and 'ring structure', where each game client is connected to its nearest and next nearest neighbors in a ring (see Figure 2).

In each round of the experiment a participant is paired with another participant, that is a neighbor in the chosen network topology. This pairing is directed by the game server, which sends each game client the JID of its partner. This is implemented by sending an XMPP message with `subject = 'new_round'` and with a message body that is the JID of the partner (see Figure 3).

In principle the two partners can now exchange messages directly through the XMPP server, but since some messages have relevance for the gameplay's logic they must also be sent to the game server. Thus, these messages must either be sent both to the partner's game client and to the game server, or simply be routed through the game server. The latter method was chosen, and is illustrated by the call diagram in Figure 3, which shows some of the messages exchanged between two game clients C_1 and C_2 via the game server, within a single round of the game. The round is started when the game server sends the `new_round` message to all clients. Chat messages are sent by a call to the `send_message()` method in NetworkingClient with `subject = 'chat'` and the chat message as the body of the message. When a player clicks the keyboard a message with `subject = 'tone'` is sent to the game server which replies back to the originating game client with a `'tone_ack'` message and sends a `subject = 'tone'` message to the game client of the partner. This causes the clicked key on both partners' game clients to briefly change color to indicate that the key (identified in the body of the message) was pressed (yellow if the player pressed, and red if the partner pressed).

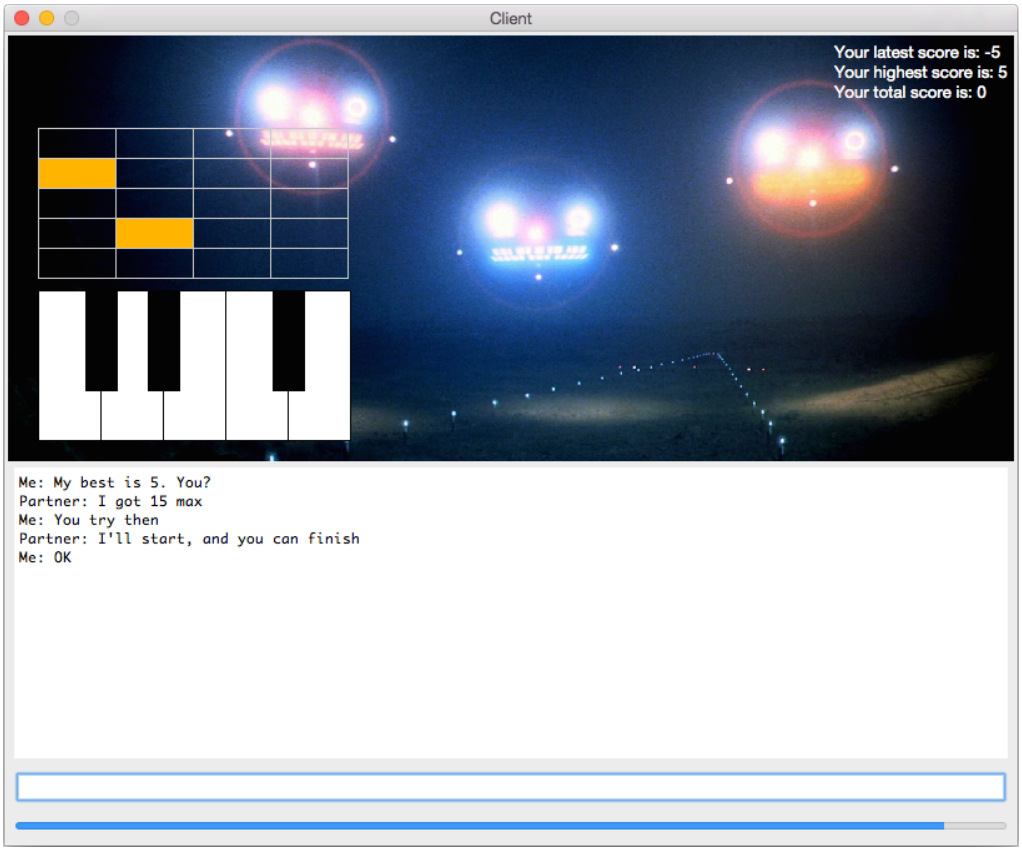

Figure 1: Screenshot of the game client from a test session. In the actual experiment the client application runs in full screen mode. The upper half contains the shared keyboard and some graphics that relates to the story behind the game. When one player presses a key, that key is briefly highlighted on the player's own and the partner's keyboard, the tone is played as audio on both client computers, and the corresponding cell in the solution grid above the keyboard is filled to give the players a visual representation of their shared melody. The lower half of the screen is a chat interface, where the two players can communicate. Below the chat input box a progress bar serves as a timer that indicates how much of the current round has passed. When a new round starts, players will be paired with a partner anew and the timer restarts, while the solution grid and the chat are cleared, so only the latest score, highest score and total (cumulative) score in the upper right corner are preserved.

Routing all messages through the game server also makes it easy to resolve contention issues. An example of this is shown in the lower part of Figure 3, where the two partners have played three of the four tones required for a melody, and they both press a fourth tone on the keyboard. In this case only one of the tones can be played and added to the melody. Since both messages (with `subject = 'tone'`) are sent to the game server, the server replies to the message it first receives (from C_2) with an acknowledgement (`subject = 'tone_ack'`) and relays this 'tone' message to the other partner (C_1). This partner's 'tone' message which is received by the game server after the first 'tone' message is simply ignored. When a pair of players have constructed a melody of four tones, the game server sends the score obtained with the melody to the game clients, and when the round ends the score will be displayed to the players.

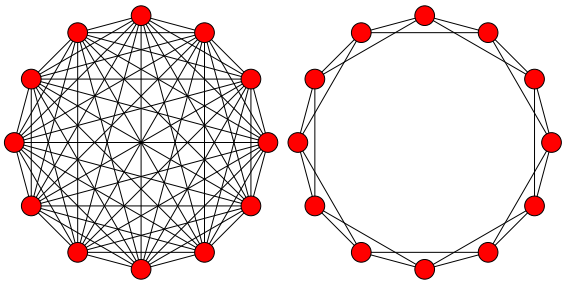

Figure 2: Examples of the two network topologies used in the experiment, for $N = 12$. To the left is the fully connected topology, where all clients are connected; and to the right the ring topology, where each client has four neighbors.

Routing all messages through the game server has the additional advantage that all messages can be time-stamped and logged centrally on the server. This makes it easy to record all the relevant data from an

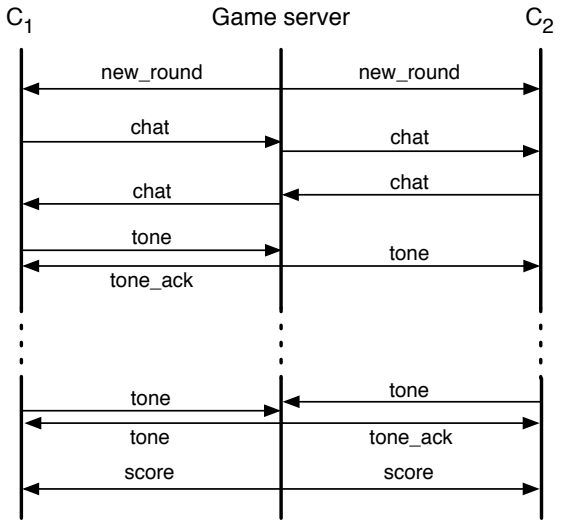

Figure 3: Call diagram illustrating the communication between two game clients, C_1 and C_2, and the game server. All messages represent calls to the `send_message()` method in NetworkingClient, and are routed through the XMPP server which is not shown. The last `tone` message from C_1 is ignored, because the pair have no more tones left to play. See main text for further details.

experimental session for later analysis. On the other hand this also puts a greater load on the game server, which may result in increased latency and jitter of the messages.

3.3 Tests and data acquisition

The solution, i.e. the NetworkingClient class, and the applications built using it can, to a large extent, be tested using unit testing. With an XMPP server on the same computer where the solution is developed the loopback interface can be used to test the exchange of messages. But an integrated test of the software for running a networked behavioral experiment ultimately has to be performed with several client computers, running on a real physical network. Testing with multiple game clients requires several people, and is an important step before running the experiment with actual participants. But this type of test is expensive in terms of people-hours, and during the early stages of the development process it may slow down development if a test requires several persons to. For this reason a stripped down game client without the need for user input is an ideal test tool. Such a 'robot client' can emulate a player using the real game client.

For the networked problem solving experiment described here, such a robot client was developed, which simply 'clicked' a random key on the keyboard at random intervals and sent chat messages of random strings with differing lengths at random intervals. Both the intervals and the string lengths were drawn from uniform distributions on an interval. The intervals lengths are parameters that can be tuned either to resemble real users, or to provide different levels of message traffic and hence load on the network, on the XMPP server, and on the game server. A screenshot of the game server during testing with 16 robot clients is shown in Figure 4. These tests showed that the game server message handler could be a bottleneck, which under sufficiently high load resulted in an increase in reply times from the server logic that affected gameplay.

Several errors were found using robot client testing, which included load tests with high message rate and stability tests with long duration. Some errors, however, were only provoked when the game client was used, and pilot experiments with human participants also resulted in new issues. After several rounds of testing using this hierarchy—unit tests, integration tests with robot clients, integration tests with human assistants, and pilot tests with human participants—the solution and the application using it are stable enough that data acquisition has started and at the time of writing data are being gathered in the lab with groups of 16 participants per experimental session.

4 CONCLUSIONS AND FUTURE WORK

In this paper a Python interface to XMPP called NetworkingClient was presented, and it was described how the class can be used to develop a real-time multiplayer experiment, only requiring access to an XMPP server, where a number of accounts can be created.

An illustrative example of an actual experiment was given, and performance was found to be satisfactory under normal conditions, although a load test with robot clients showed that the game server could cause unacceptable delays under high load, indicating that the present version will not scale to a large number of users.

In view of recently developed solutions described in section 1.1 it is clear that other mature alternatives exist, that would achieve most of the design goals listed for the solution presented here. The present solution has an advantage in cases where an existing Python implementation of the client interface exists. For example the example experiment presented here already existed in a single-player version implemented for PsychoPy and large pars of the code could therefore be reused, although the multiplayer version was written using the TkInter GUI library instead of Psy-

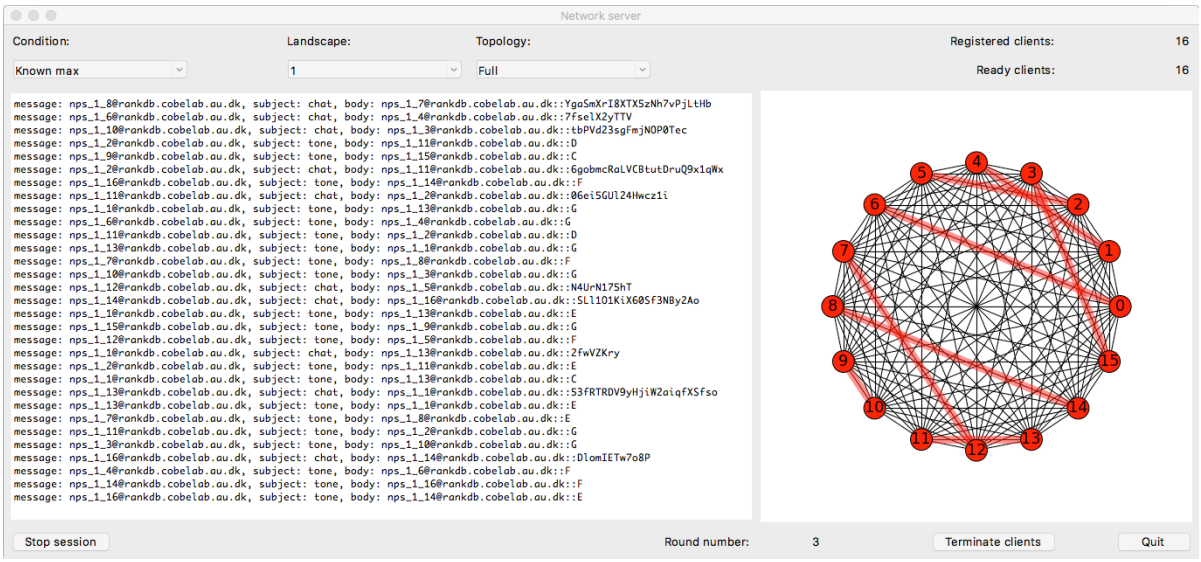

Figure 4: Screenshot of the game server from a test session with 16 robot clients. Messages from the clients are shown continuously in the left side of the window. In each game round a the game server constructs a new set of pairs drawn at random. This is shown on the right side of the window, where red lines indicate the clients that are paired in the current game round, and black lines indicate which clients are neighbors. The example illustrated here is for the fully connected topology, where all clients are connected to each other.

choPy. Another instance where the present solution is more appropriate than the alternatives, is when the client has to interface with peripheral equipment. This is an area where PsychoPy excels, and where NetworkingClient can be used to connect several PsychoPy clients through a game server.

4.1 Future work

The NetworkingClient solution has so far only been used for research projects at Cognition and Behavior Lab. Continued development will be needed to optimize performance. The game server message handler was identified as a potential bottleneck, and this can possibly be alleviated by providing a method to register the message handler using the hooks in the SleekXMPP library rather than creating a message queue.

Another topic for future work is to integrate NetworkingClient with PsychoPy to make it as easy as possible to create real-time multiuser experiments with PsychoPy.

ACKNOWLEDGEMENTS

This project has been supported by seed funding 2016 from the Interacting Minds Centre, Aarhus University. The author wishes to thank René Frederiksen for working on the initial implementation of one of the experiments and the XMPP client code. The game client described in the example application was based on a single-player version programmed in PsychoPy by Kristian Tylén.

REFERENCES

Albano, M., Ferreira, L. L., Pinho, L. M., and Alkhawaja, A. R. (2015). Message-oriented middleware for smart grids. *Computer Standards & Interfaces*, 38:133–143.

Balietti, S. (2016). nodeGame: Real-time, synchronous, online experiments in the browser. *Behavior Research Methods*, pages 1–20.

Barkoczi, D. and Galesic, M. (2016). Social learning strategies modify the effect of network structure on group performance. *Nature Communications*, 7:13109.

Baronchelli, A., Ferrer-i Cancho, R., Pastor-Satorras, R., Chater, N., and Christiansen, M. H. (2013). Networks in Cognitive Science. *Trends in Cognitive Sciences*, 17(7):348–360.

Bernstein, D., Ludvigson, E., Sankar, K., Diamond, S., and Morrow, M. (2009). Blueprint for the Intercloud - Protocols and Formats for Cloud Computing Interoperability. In *2009 Fourth International Conference on Internet and Web Applications and Services*, pages 328–336.

Caminiti, S., Cicali, C., Gravino, P., Loreto, V., Servedio, V. D. P., Sirbu, A., and Tria, F. (2013). XTribe: A Web-Based Social Computation Platform. In *2013 International Conference on Cloud and Green Computing*, pages 397–403.

Centola, D. and Baronchelli, A. (2015). The spontaneous emergence of conventions: An experimental study of cultural evolution. *Proceedings of the National Academy of Sciences*, 112(7):1989–1994.

Chen, D. L., Schonger, M., and Wickens, C. (2016). oTree—An open-source platform for laboratory, online, and field experiments. *Journal of Behavioral and Experimental Finance*, 9:88–97.

Claypool, M. and Claypool, K. (2006). Latency and Player Actions in Online Games. *Commun. ACM*, 49(11):40–45.

Dustdar, S., Nastic, S., and Scekic, O. (2016). A Novel Vision of Cyber-Human Smart City. In *2016 Fourth IEEE Workshop on Hot Topics in Web Systems and Technologies (HotWeb)*, pages 42–47.

Fischbacher, U. (2007). z-Tree: Zurich toolbox for ready-made economic experiments. *Experimental Economics*, 10(2):171–178.

Fritz, N. and Stout, L. (2016). SleekXMPP (https://github.com/fritzy/SleekXMPP).

Fusaroli, R. and Tylén, K. (2016). Investigating Conversational Dynamics: Interactive Alignment, Interpersonal Synergy, and Collective Task Performance. *Cognitive Science*, 40(1):145–171.

Granovetter, M. S. (1973). The Strength of Weak Ties. *American Journal of Sociology*, 78(6):1360–1380.

Hannay, J. E., MacLeod, C., Singer, J., Langtangen, H. P., Pfahl, D., and Wilson, G. (2009). How do scientists develop and use scientific software? In *2009 ICSE Workshop on Software Engineering for Computational Science and Engineering*, pages 1–8.

Hawkins, R. X. D. (2015). Conducting real-time multiplayer experiments on the web. *Behavior Research Methods*, 47(4):966–976.

Hawkins, R. X. D. and Goldstone, R. L. (2016). The Formation of Social Conventions in Real-Time Environments. *PLOS ONE*, 11(3):e0151670.

Hornsby, A. and Walsh, R. (2010). From instant messaging to cloud computing, an XMPP review. In *IEEE International Symposium on Consumer Electronics (ISCE 2010)*, pages 1–6.

Lee, N. (2004). Jabber for Multiplayer Flash Games. *Comput. Entertain.*, 2(4):13–13.

Markopoulou, A. P., Tobagi, F. A., and Karam, M. J. (2002). Assessment of VoIP quality over Internet backbones. In *Proceedings.Twenty-First Annual Joint Conference of the IEEE Computer and Communications Societies*, volume 1, pages 150–159 vol.1.

Mønster, D., Håkonsson, D. D., Eskildsen, J. K., and Wallot, S. (2016). Physiological evidence of interpersonal dynamics in a cooperative production task. *Physiology & Behavior*, 156:24–34.

Nezhdanov, A. and Rasmussen, N. (2013). xmpppy: the jabber python project (https://sourceforge.net/projects/xmpppy/).

Peirce, J. W. (2007). PsychoPy—Psychophysics software in Python. *Journal of Neuroscience Methods*, 162(1–2):8–13.

Peirce, J. W. (2009). Generating stimuli for neuroscience using PsychoPy. *Frontiers in Neuroinformatics*, 2.

Saint-Andre, P. (2011a). Extensible Messaging and Presence Protocol (XMPP): Core (IETF RFC 6120).

Saint-Andre, P. (2011b). Extensible Messaging and Presence Protocol (XMPP): Instant Messaging and Presence (IETF RFC 6121).

Schuster, D., Grubitzsch, P., Renzel, D., Koren, I., Klauck, R., and Kirsche, M. (2014). Global-Scale Federated Access to Smart Objects Using XMPP. In *2014 IEEE International Conference on Internet of Things (iThings), and IEEE Green Computing and Communications (GreenCom) and IEEE Cyber, Physical and Social Computing (CPSCom)*, pages 185–192.

Wallot, S., Mitkidis, P., McGraw, J. J., and Roepstorff, A. (2016). Beyond Synchrony: Joint Action in a Complex Production Task Reveals Beneficial Effects of Decreased Interpersonal Synchrony. *PLOS ONE*, 11(12):e0168306.

Wilson, G. (2016). Software Carpentry: lessons learned. *F1000Research*, 3(62).

Woods, A. T., Velasco, C., Levitan, C. A., Wan, X., and Spence, C. (2015). Conducting perception research over the internet: a tutorial review. *PeerJ*, 3:e1058.