university of groningen / faculty of science and engineering / bernoulli institute

Tools for Gossip

Bachelor's Project Thesis

R.A. Meffert, s2702207, r.a.meffert@student.rug.nl
Supervisor: Dr. B.R.M. Gattinger

Abstract: The field of gossip—concerned with the way information spreads in networks of agents—has been the subject of research since at least 1927. With the introduction of new paradigms within this field, such as dynamic and distributed gossip, the field has become more complex. Information on dynamic (distributed) gossip is available in academic literature, but the protocols used and their consequences are not easily understood intuitively. To improve ease of understanding, we present a tool for visualising gossip graphs and working with arbitrary gossip protocols. The goal of the tool is to allow students and researchers to more quickly gain an understanding of the most important aspects of gossip. To evaluate this goal, a survey was carried out among field experts. Respondents reported they find the tool easy to use and consider it useful in a study setting. Some users also considered the tool useful in a research setting. Furthermore, feedback from the survey was used to improve the user experience and gain an understanding of which features are the most important to users.

1 Introduction

Gossip protocols are protocols that describe the way rumours—or, more generally, pieces of information—are shared in multi-agent environments. The goal of the protocols is to communicate all pieces of information to all agents. A lot of research has been done in this field, starting with research on the spread of infectious diseases (Kermack & McKendrick, 1927).

The origin of the gossip problem has proven to be hard to pinpoint exactly. Most papers cite an unpublished work by R. Chesters and S. Silverman at the University of Witwatersrand. This is most likely based on a review (Graham, 1973) of a paper by Tijdeman (1971). Tijdeman claims the problem was first formulated by A. Boyd, but Graham claims Chesters and Silverman proposed the original problem. Another possibility is given by Lebensold (1973), who claims the problem was proposed by Erdős. Although the paper by Tijdeman seems to be considered one of the first solutions to the gossip problem by many, an unpublished paper by R.T. Bumby and J. Spencer might be earlier. Harary and Schwenk (1974) cite this paper as being referenced in a work from 1966.

In short, agents are represented as nodes in a graph, with the edges representing a relation. This relation can be the number relation N, representing that one agent has the "phone number" of another agent, and the secret relation S, representing that one agent knows the secret of another agent. Many authors (e.g., Baker & Shostak, 1972; Hajnal et al., 1972; Lebensold, 1973; Tijdeman, 1971) have proven that it takes no fewer than $2n - 4$ calls before all agents know all secrets, where n is the number of agents, given that all agents know the numbers of all other agents—that is, the number relation N is *complete*.

The problem as formulated above requires the oversight of a central authority in order to know whether all agents know all secrets. However, there are many applications where this is not feasible or desirable as this becomes very computationally expensive as the number of agents increases (Kermarrec & van Steen, 2007). Another problem is that in real-world applications, often the agents do not know the phone numbers of all other agents. Therefore, it is useful that agents not only spread their secrets, but also the phone numbers of other agents. This has led to the sub-fields of *distributed gossip*, addressing the first issue, and *dynamic gossip*, addressing the second. The combination of these fields, where there is no overseer and the number relation can change during runtime, is called *distributed dynamic gossip*.

This paper will describe a tool to make it easier to explore distributed dynamic gossip. It is able to visualise the connections between agents, allows exploring the execution tree of different protocols and validate call sequences given a graph and a protocol.

1.1 Earlier work

1.1.1 Applications of gossip

Gossip has, since its first formal description, found many different applications. One of the earlier applications we could find was used for database consistency in the Xerox Corporate Internet (Demers et al., 1988). More recent applications in

system consistency can be found in the applications in Amazon's DynamoDB software (DeCandia et al., 2007) and the SWIM protocol (Das et al., 2002). However, more applications than database consistency exist: Notable examples include its use in genome research (Liben-Nowell, 2002), alternatives to the proof-of-work in blockchain (Baird, 2016; Renesse, 2016) and ad-hoc network routing (Haas et al., 2006).

1.1.2 Visualisation tools

A researcher or student interested in dynamic gossip will probably start looking for some way to gain an intuitive understanding of the field. At the time of writing, a couple of applications that demonstrate properties of dynamic gossip are available. However, most of these require either an established understanding of the field and/or proficiency using command-line tools.

The first two tools are by Gattinger (2016, 2017): one works in the browser (Gattinger, 2016), the other is a Haskell library (Gattinger, 2017). The first tool, `WebGossip`, is able to visualise gossip graphs based on user input. While the input format for that tool is different from the one used in this tool (for an explanation of the format for our tool, see Section 2.1.1), it is similar in that it also takes text input and displays the gossip graph graphically. The tool also generates code for use in LaTeX papers

The second tool, `GoMoChe`, has more features but, being a Haskell library, is not as easy to use. The main differences between these two tools and our tool can be seen in Table 1.1

Two other tools worth mentioning are by Maffre (2016) and Moelker (2016). The first allows users to generate domain and problem files in the Planning Domain Definition Language (PDDL) for a given gossip protocol, making it possible to analyse protocol execution using PDDL software. The second is a visualisation tool, but due to the lack of a description on the project page, it is not entirely clear what the tool does. It seems to implement some specific gossip protocol and simulate interactions between agents.

In conclusion, very few tools are available for exploring dynamic gossip. The tools that *are* available are either lacking in features or provide an uninviting user interface. Our tool aims to improve on both of these aspects.

1.2 Notation

This paper takes its notation from Van Ditmarsch et al. (2018) and Van Ditmarsch et al. (2019). The notation used is explained below.

Definition 1 (Gossip graph)**.** Let A be a set of agents $\{a, b, \dots\}$. Two directed binary relations on A are defined: $N, S \subseteq A^2$. The first denotes the *number* relation, the second the *secret* relation. A gossip graph G is then defined as a triple (A, N, S).

Definition 2 (Relations on gossip graphs)**.**

Definition 2.1 (Binary relation)**.** When agent x has relation P to agent y, this is denoted as Pxy, which is a shorthand notation of $(x, y) \in P$.

Definition 2.2 (Identity relation)**.** The set of relations where all agents have a relation to themselves is denoted I_A.
Formally: $I_A := \{(x, x) \mid x \in A\}$

Definition 2.3 (Converse relation)**.** The set of reverse relations in P is denoted P^{-1}.
Formally: $P^{-1} := \{(x, y) \mid Pyx\}$

Definition 2.4 (Composition relation)**.** The composition of the relations P and Q is a new relation such that the tuple (x, z) is in said new relation if and only if there exists another agent y such that $(x, y) \in P$ and $(y, z) \in Q$.
Formally: $P \circ Q := \{(x, z) \mid \exists y((x, y) \in P \wedge (y, z) \in Q)\}$

Definition 2.5. The set of agents x has relation P with is denoted P_x.
Formally: $P_x := \{y \in A \mid Pxy\}$

Definition 2.6. The ith composition of relation P with itself is denoted P^i.
Formally:

$$P^i := \begin{cases} P & \text{for } i = 1 \\ P^{i-1} \circ P & \text{for } i > 1 \end{cases}$$

Definition 2.7. The transitive closure of P is denoted P^*.
Formally:

$$P^* := \bigcup_{i \geq 1} P^i$$

Definition 3 (Properties of gossip graphs)**.**

Definition 3.1 (Weakly connected)**.** A graph G is weakly connected if for all $x, y \in A$ there exists a path in N from x to y or from y to x, that is, the symmetric transitive closure of N is complete.
Formally:

$$\forall x, y \in A : (x, y) \in (N \cup N^{-1})^*$$

Definition 3.2 (Strongly connected)**.** A graph G is strongly connected if for all $x, y \in A$ there exists a path in N from x to y.
Formally:

$$\forall x, y \in A : (x, y) \in N^*$$

Table 1.1: Comparison between tools. The tools by Maffre (2016) and Moelker (2016) have not been included in this comparison since they are too different for direct comparison.

Feature	`GoMoChe`	`WebGossip`	This paper
Interface	Command line	Web	Web
Gossip graph visualisation	Yes	Yes	Yes
LaTeX output	No	Yes	Planned
Call execution	No	No	Yes
Call sequence execution	No	No	Yes
Show allowed calls	No	No	Yes
Show allowed sequences	Yes	No	No
Show execution tree	No	No	Yes
Protocols	LNS, CMO[1], CMOwLoG, PIG, ANY	N/A	LNS, CO, wCO, SPI, ANY, TOK, *custom*[2]

[1] Call-me-once; called CO in this paper.
[2] Custom protocols were planned, but not finished in the implementation for this paper.

Definition 3.3 (Sun graph)**.** A graph G is a sun graph iff its non-terminal nodes are strongly connected.

Definition 4 (Calls)**.** A call from an agent x to an agent y is denoted xy.

Definition 4.1 (Call sequence)**.** Call sequences are denoted using lowercase greek letters, starting with σ. The empty sequence is denoted ϵ and $\sigma\,;\tau$ denotes the concatenation of two sequences.

Definition 4.2 (Subsequence)**.** The set of all calls in a sequence σ containing x is denoted σ_x.

Formally:

$$\sigma_x := \begin{cases} \epsilon & \text{if } \sigma = \epsilon \\ \tau_x\,;uv & \text{if } \sigma = \tau\,;uv \wedge (x = u \vee x = v) \\ \tau_x & \text{if } \sigma = \tau\,;uv \wedge \neg(x = u \vee x = v) \end{cases}$$

Definition 4.3 (Effect of a call on a relation)**.** The state of relation P after call xy is denoted P^{xy} and defined by: $P^{xy} := P \cup (\{(x, y), (y, x)\} \circ P)$

Definition 4.4 (Effect of a call on a gossip graph)**.** The state of gossip graph G after call xy is denoted G^{xy} and is defined by: $G^{xy} := (A, N^{xy}, S^{xy})$

Definition 4.5 (Effect of a call sequence on a gossip graph)**.** The state of gossip graph G after call sequence σ is denoted G^σ and is defined by:

$$G^\sigma := \begin{cases} G & \text{if } \sigma = \epsilon \\ (G^{xy})^\tau & \text{if } \sigma = xy\,;\tau \end{cases}$$

Definition 4.6 (Call sequence possibility)**.** A call sequence $\sigma\,;xy$ is *possible* on a gossip graph G iff xy is possible on G^σ (i.e., $N^\sigma xy$) and σ is possible on G.

Definition 4.7 (Call permissibility)**.** A call xy is *permitted* under protocol condition $\pi(x, y)$ on a gossip graph G iff $x \neq y$, Nxy and $G \models \pi(x, y)$.

Definition 4.8 (Call sequence permissibility)**.** A call sequence $\sigma\,;xy$ is permitted on a gossip graph G iff xy is permitted on G^σ and σ is permitted on G.

Definition 5 (Gossip Protocols)**.**

Definition 5.1 (Protocol condition)**.** $\pi(x, y)$ represents a protocol condition, that is, a condition that needs to be satisfied before a call can be made. This report takes this condition, like Van Ditmarsch et al. (2018), as a boolean combination of the following constituents: $S^\sigma xy$, $xy \in \sigma_x$, $yx \in \sigma_x$, $\sigma_x = \epsilon$, $\sigma_x = t\,;xz$ and $\sigma_x = t\,;zx$.

Definition 5.2 (Gossip Protocol)**.** A gossip protocol P is a protocol condition $\pi(x, y)$ together with a non-deterministic algorithm of the following form:

> Until all agents are experts, select $x, y \in A$, such that $x \neq y$, Nxy, and $\pi(x, y)$, and execute call xy.
>
> (van Ditmarsch et al., 2018, Def. 5).

2 Method

To provide as much useful information as possible, the tool needs to provide information on a number of aspects of gossip. The first aspect that will be discussed is gossip graph visualisation (Section 2.1.1), followed by protocol execution (Section 2.1.2), call sequence validation and call (sequence) execution. First, a reasoning will be given behind the programming language of choice. Then, these aspects are discussed in terms of their meaning in both the context of gossip theory and of their implementation. Lastly, some time will be spent to explain how these aspects are integrated into the application.

2.1 Implementation

This project uses Elm, a statically typed functional programming language for web development with its roots in Functional Reactive Programming (Czaplicki & Chong, 2013). This has several advantages:

1. Implementing mathematical functions is more natural because functions in Elm are pure;
2. Elm is compiled to standard Javascript. Since all modern browsers support Javascript, this ensures the application is cross-platform;
3. Elm does static type checking while compiling, ensuring type safety and no runtime exceptions;

Another advantage of using a functional language is that much of the notation introduced in section 1.2 can be translated fairly directly. For example, to evaluate some of the protocol conditions, the last call in a call sequence needs to be checked. In mathematical notation this is represented as $\sigma_x = \tau\,;xy$ ("the last call in the sequence of calls containing x was a call made by x to another agent y"). This can be represented in Elm quite naturally, as can be seen in Listing 1.

Listing 1: $\sigma_x = \tau\,;xy$ in Elm.

```
lastFrom agent sigma_x =
    case reverse sigma_x of
        [] ->
            False

        (x, y) :: tau ->
            x == agent
```

This states that if $\sigma_x = \epsilon$, the condition is false. Otherwise, it checks to see if the last call was made by x, and returns true if that is the case.

The source code of the tool can be found at `https://github.com/ramonmeffert/tools-for-gossip`. The tool is available for use at `https://ramonmeffert.github.io/tools-for-gossip`.

2.1.1 Gossip graph visualisation

One of the easiest ways to visualise connections in a network is by using a graph. Therefore, the application is able to visualise gossip graphs based on user input. The input is provided by users in text, and uses a format based on the one used in the appendix of van Ditmarsch et al. (2019). Since no complete grammar of that format is given, the format has been adapted to be a bit more permitting than could be expected based purely on its description. For example, our implementation considers the inputs `Abc aBc abC` and `Abc Bac Cab` the same, while the original format (arguably) would only accept the first input.

The basic idea behind the format is that every agent is represented by a letter segment. The letters in this segment represent the relations of the agent represented by that segment. Uppercase letters represent the secret relation S and lowercase letters represent the number relation N. Because every agent knows their own secret ($I_A \subseteq S$), it is possible to find names for all agents if and only if the input represents a valid gossip graph. That is, every agent is represented by some letter segment such that the number of segments is equal to the number of unique agent names, and every letter segment contains a secret relation that allows it to be uniquely identified. This procedure is described in more detail in Algorithm 2.1.

Once the agent names have been found, it is possible to parse the relations from the input string. Algorithm 2.1 assumes the first agent name it finds represents the first agent. Thus, the first segment represents the relations of that agent.

This interpretation of the format is likely a bit more permissive than the original format. For example, in the original format, the names of agents always follow alphabetic order – agent 1 is A, agent 2 is B, et cetera. Furthermore, the text segments in the original format are always ordered alphabetically. These requirements are not present in the implementation, so an input like `Xyz Yzx Zxy` is allowed. However, the tool does generate a "canonical string representation" which does follow the original rules.

2.1.2 Executing calls

Single calls When a gossip graph is given by the user, the tool allows them to execute calls and call sequences. Because the possible calls depend on the active protocol, a list of possible calls is given for the currently active protocol (Figure 2.3). The possible calls are found by checking for call permissibility as defined in Definition 4.7. When the user clicks one of these calls, the call is executed. The call then happens according to Definition 4.4.

Call sequences It is also possible to execute entire call sequences. The user is able to enter a call sequence using a text input according to the format given in Definition 4.1 (Figure 2.2). This sequence is then checked for possibility (Definition 4.6) and permissibility (Definition 4.8). It should be noted that these checks are done in the order mentioned here. This means that the user receives feedback on their input when they enter a call sequence that is not possible on the current gossip graph. This has the added advantage that the system need not check for permissibility if the user enters a call

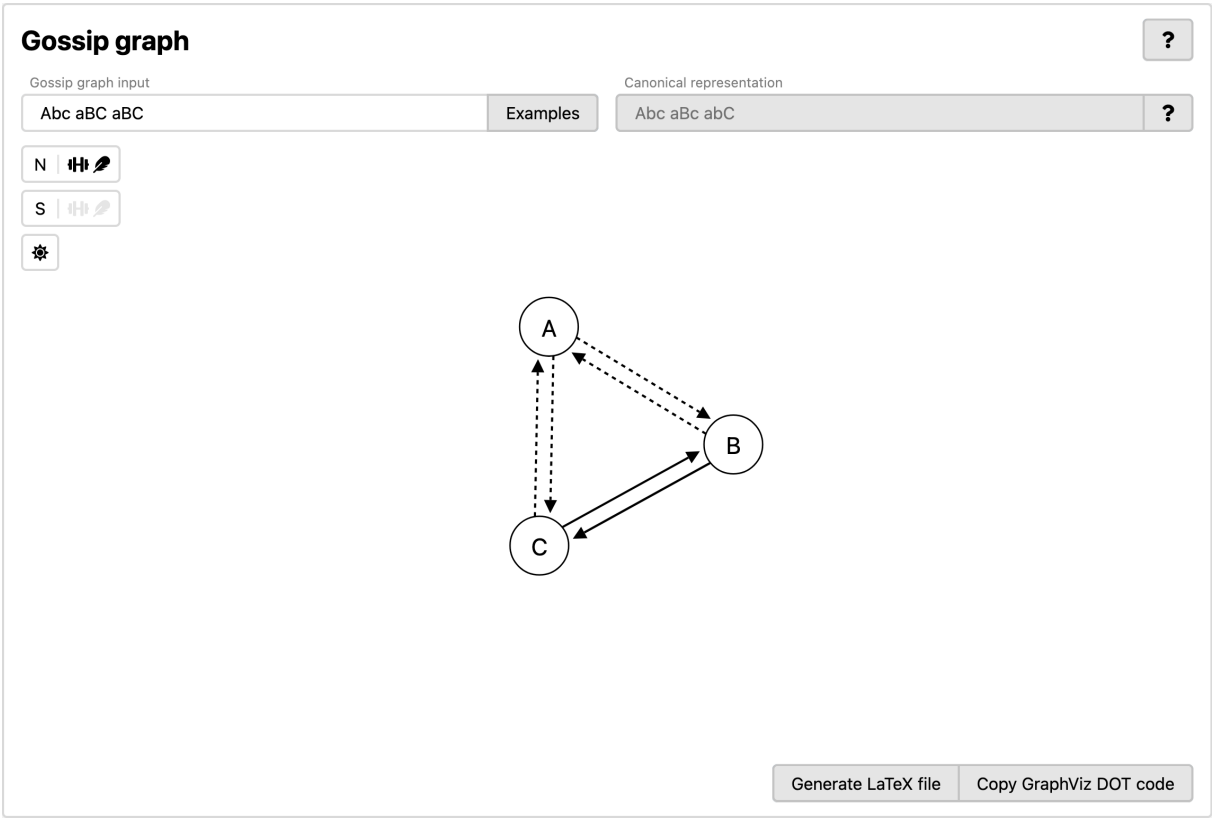


Figure 2.1: A visualised gossip graph. Users enter the text representation of the gossip graph on the left. The icons below this input field represent whether the number relation (N) and secret relation (S) are weakly (feather icon, Definition 3.1) and/or strongly (dumbbell icon, Definition 3.2) connected. The sun icon represents whether the graph is a sun graph (Definition 3.3).

Algorithm 2.1: Finding agent names.

input : A list of letter sequences called segments
output : A list of agent names

```
1  names ← {}
2  foreach segment ∈ segments do
3      possibleNames ← GetUppercaseLetters(segment)
4      UniqueNameFound ← false
5      foreach name ∈ possibleNames do
6          if name ∉ names then
7              names ← {name} ∪ names
8              UniqueNameFound ← true
9          end
10     end
11     if ¬UniqueNameFound then
12         ShowError()
13     end
14 end
15 return names
```

Figure 2.2: The interface for evaluating and executing call sequences. The check indicates the entered call sequence is possible and permissible.

sequence that is impossible. Additionally, because possibility is a requirement for permissibility and its check is executed before the permissibility check, this check does not need to occur in the check for permissibility; if the system reaches the permissibility check, the call sequence is guaranteed to be possible.

Existing protocols The tool by default includes the gossip protocols mentioned in van Ditmarsch et al. (2018). These are *Any* (`ANY`), *Call Once* (`CO`), *Weak Call Once* (`wCO`), *Spider* (`SPI`), *Token* (`TOK`) and *Learn New Secrets* (`LNS`).

Custom protocols To allow more flexibility in using protocols, our tool allows the user to create their own gossip protocols.* This is done through a drag-and-drop interface, in which users can arrange boolean constituents or groups of boolean constituents. These constituents are then interspersed with either logical conjunction ($\wedge$) or logical disjunction ($\vee$) (see also Figure 2.3).

The creation of custom protocols is based on the language $\mathcal{L}_B(V)$ of boolean formulas, given by the Backus-Naur form: $\varphi ::= \top \mid p \mid \neg\varphi \mid \varphi \wedge \varphi$, where $p \in V$. This is extended with the logical disjunction by virtue of the formula $\varphi \vee \psi := \neg(\neg\varphi \wedge \neg\psi)$. For our tool, the vocabulary V is the set of constituents of protocol conditions as defined in Definition 5.1.

The implementation of this functionality is based on a binary tree in which nodes can either represent constituents or connectives ($\vee$ and $\wedge$). An example of this can be seen in Figure 2.4. Users are able to modify the by dragging and dropping constituents to new locations, toggling connectives between ($\wedge$) and ($\vee$) by clicking them, and deleting constituents. The application will then parse this into an Elm function which can be used to determine the possible calls.

*Note: at the time of publication, this part of the tool was not finished.

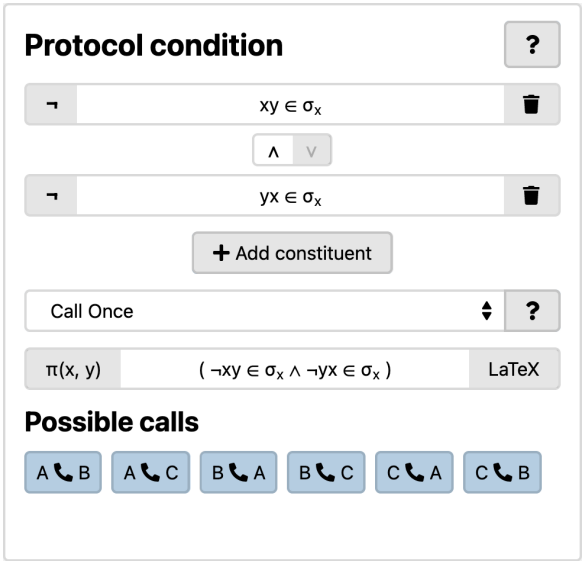
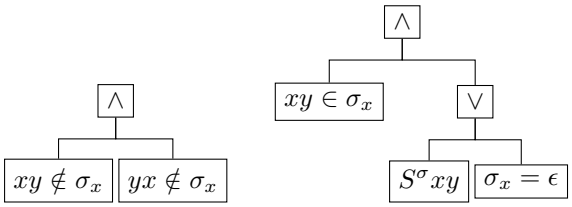


Figure 2.3: The proposed interface for creating custom gossip protocols.



(a) The *Call me Once* protocol as visible in Figure 2.3.

(b) A custom protocol ($xy \in \sigma_x \wedge (S^\sigma xy \vee \sigma_x = \epsilon)$).

Figure 2.4: The binary tree representation of protocol conditions.

2.1.3 Execution tree

Whenever a call or call sequence is executed, the calls are recorded on a timeline. This timeline is displayed as the initial graph and the sequence of calls performed on it. Users are able to navigate between different states of the gossip graph by clicking a call, which changes the state of the gossip graph to the state it had after the clicked call.

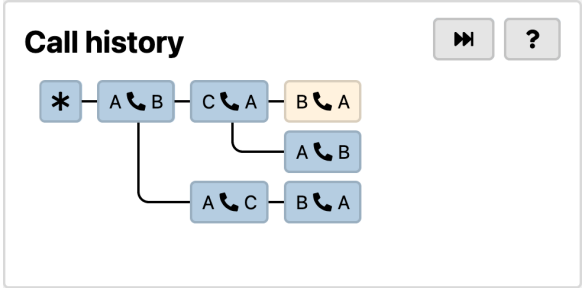


Figure 2.5: The way the execution tree is visualised in our tool. The $*$ node represents the initial gossip graph. The yellow node represents the current 'location' in the execution tree.

There are two ways in which the execution tree can be generated. The first is by executing calls or call sequences. This appends the call (or call sequence) to the current state of the gossip graph. This allows for branching in the execution tree: if calls have already been executed after the current state, a new branch will be added. The second is by using the 'forward' button (visible in Figure 2.5, the leftmost button in the top right corner). Pressing this button will present the user with a dialog in which they can determine the maximal branching depth of the execution tree, allowing a depth of at most 5 levels. This maximum has been implemented in order to make sure the program can calculate the execution tree, as the number of nodes to be calculated grows exponentially. Circumstantial evidence suggests the tool can handle a depth of 5 under the `ANY` protocol given an initial graph $G = (A = \{a, b, c\}, N = A^2, S = I_A)$ (resulting in $6^5 = 7,776$ nodes) without much difficulty.

2.1.4 User interaction and support

Error messages In order to make the tool more user-friendly, it provides error messages explaining the problem when an error occurs. An example of this can be seen in Figure 2.6.

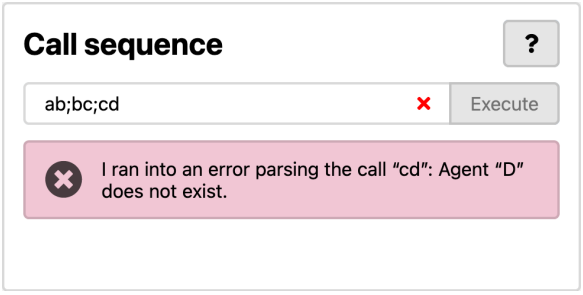


Figure 2.6: An error that is shown when a call sequence containing a non-existing agent is entered.

Documentation To make the documentation as accessible as possible, the user documentation was integrated in the user interface. When users click the `[?]` button that is present on every section, they are shown a popup window containing an explanation of that section's functionality. An example of this can be seen in Figure 2.7

2.2 Evaluation

To evaluate the usability and applicability of the tool, a short exploratory survey was sent out to a number of researchers active in the field of (dynamic) gossip as well as students of the master course "Design of Multi-Agent Systems" at the University of Groningen. The purpose of this survey

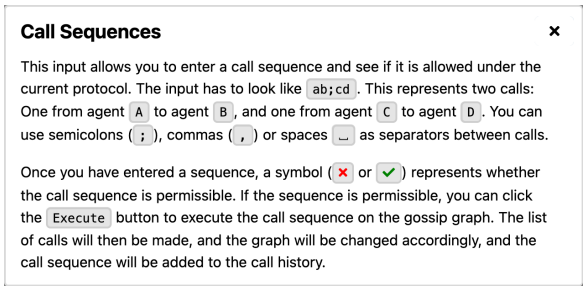


Figure 2.7: The documentation message for call sequences.

was to evaluate the ease of use and applicability of the tool, to identify most important features and to gather suggestions for improvement and/or extensions. The questions used can be found in Appendix A.

The response types for this survey are semantic differential scales, Yes/No/Maybe questions, multiple-choice lists and open-input questions. The semantic differential scales will be analysed using a Wilcoxon Rank Sum test (sometimes also known as a Mann-Whitney U-test). The responses will be treated as an interval scale centered around the neutral response, which is at 3. For the Yes/No/-Maybe questions, a cross table will be used. Lastly, the open-answer questions will be paraphrased in the analysis; they are intended to gather possible improvements and do not contribute to the statistical analysis of the tool.

3 Results

The survey was open for responses during the period between December 19, 2020 and January 13, 2021 and received 12 responses (N = 12). A Wilcoxon Rank Sum test showed a significant non-neutral response on questions about familiarity with gossip (median = 4, V = 45, p = 0.007), interface design (median = 4.5, V = 78, p = 0.002) and usefullness of error messages (median = 4, V = 74, p = 0.005). Users did not require significantly low or high amounts of time before they understood what each part of the interface was for (median = 3, V = 6, p = 0.19). The majority of users thought the tool could be useful in a study setting, but the response for a research setting was more divided (Table 3.1).

Question 8 (Importance of features) resulted in the following order of importance (the number between parentheses represents the number of responses):

1 (10) Checking permissibility of call sequences on gossip graphs under certain protocols

(10) Visualising gossip graphs

Table 3.1: Usefulness of the tool in a research or study setting.

	Response		
Setting	No	Maybe	Yes
Research	1 (8%)	5 (33%)	6 (50%)
Study	0 (0%)	0 (0%)	12 (100%)

2 (9) Executing call sequences on gossip graphs

3 (8) Exporting gossip graphs to LaTeX or GraphViz formats

4 (6) Creating custom protocols by creating a boolean combination of constituents (as defined in the paper "Dynamic Gossip" by van Ditmarsch et al. (2019))

(6) Gossip graph statistics: whether a graph is a sun graph, and whether a relation is strongly/weakly connected

5 (5) Exporting execution tree diagrams to LaTeX or GraphViz formats

6 (4) Generating execution trees for gossip protocols

(4) Navigating between different states of a gossip graphs after (a) call(s) have been made

4 Discussion

4.1 Survey

Respondents evaluated the tool positively on both the design and usefullness of error messages. Respondents did not take significantly little or much time to understand the user interface. Though this result can be interpreted positively, it also leaves room for improvement. Speed of understanding could be improved by, for example, adding an interactive tutorial of the tool the first time a user loads the web page.

Furthermore, while the tool was considered useful in a study setting, only about half of the respondents considered the tool to be useful in a research setting. Extending the tool with more types of gossip (as will be discussed in section 4.2.1), as well as finishing the custom protocol creation could improve the usability in a research setting.

While the survey yielded useful results, its exploratory nature along with the low number of respondents mean that its results are difficult to analyse quantitatively. Therefore, it could be interesting to perform a larger user study. This could include a larger survey focused more on objective analysis of the tool's ergonomics. Another option would be to perform a number of cooperative walkthroughs to gain more insight in problems users might face.

4.2 Further research

At the time of publication of this report, several planned features – some part of the initial concept, some arising from the results of the survey – have not been implemented yet. For completeness' sake, we will list these features here shortly:

- The custom protocol creation (Section 2.1.2) does not work yet;
- The rendered graph sometimes jumps when changing it;
- User input is not persisted; when the page is refreshed, all input is gone.

Besides these features, a more up-to-date list of smaller planned features and bug fixes can be found at `https://github.com/RamonMeffert/tools-for-gossip/issues`.

4.2.1 Possible extensions of the tool

The current version of the tool allows users to model dynamic gossip protocols as defined in van Ditmarsch et al. (2018). However, there is more to gossip than what is described in that paper. In this section, other papers on (dynamic) gossip are presented which introduce additional concepts. To make the tool more broadly usable, it could be interesting to add some or all of these aspects to the tool.

The first possibility considered here is that of meta-knowledge or epistemic information, such as in Herzig and Maffre (2017). That means that agents can not only know secrets and numbers, but also whether other agents know a secret or number. This allows for more complex protocols. An example of this can be found in the `GoMoChe` tool (Gattinger, 2017).

Another idea is to include a temporal aspect, such as in Cooper et al. (2019) (note that this paper also includes epistemic information). This means that agents cannot always be reached. The paper gives as examples for this problem communication between satellites on different orbits that might not always be able to 'see' each other, or a phone with a battery that sometimes does not have enough charge to communicate. This might be interesting, because this means that instead of calls being either always possible or always impossible, they might *sometimes* be possible or impossible as well.

A related problem is introduced by van den Berg and Gattinger (2020): normally, it is assumed that agents in dynamic gossip are reliable and tell the

truth about the secrets they hold. However, if agents lie, many known properties of dynamic gossip suddenly no longer hold. By introducing unreliable agents, a new goal is added to the gossip system: *identify the unreliable agents*. An implementation of this (though slightly different from the version described in the paper mentioned before) can be found at van den Berg (2020).

Another modification to secrets could be taken from Demers et al. (1988), who propose a system in which all secrets have a 'hotness'. This 'hotness' indicates how often the secret has been spread. This can then be used as a heuristic to determine whether it is useful to communicate the secret. This might be interesting because in the current system, an agents communicates all their secrets in a call. However, in a real-life application, this might not be desired: In large networks, communicating all secrets all the time might lead to large amounts of data being sent even though this might be data the agents receiving the secrets already have.

Besides these improvements to the functionality of the tool, another possible improvement would be to make it suitable for usage on mobile devices such as smartphones and tablets. According to Cisco (2020), the number of mobile devices used to access the internet is larger than the number of desktop devices, and is still increasing. This increase is also most visible among younger people, and since a key user group of this tool is intended to be students, having the tool be usable on a mobile device will probably increase the likelihood of it being used.

5 Conclusion

We have described a tool that allows people interested in dynamic gossip to explore the subject interactively. The tool contains a visualisation of gossip graphs and allows users to execute calls and call sequences under several gossip protocols and observe the results. To evaluate the usefulness, a survey was sent out. Respondents evaluated the design and the presence of the error messages positively. All respondents considered the report useful in a study setting, and some also considered it useful in a research setting. Finally, we presented a number of ideas for improvements and extensions of the tool which could increase the usefulness of the tool.

References

Baird, L. (2016). The swirlds hashgraph consensus algorithm: Fair, fast, byzantine fault tolerance. *Swirlds, Inc. Technical Report SWIRLDS-TR-2016*, *1*. https://www.swirlds.com/wp-content/uploads/2016/06/2016-05-31-Swirlds-Consensus-Algorithm-TR-2016-01.pdf

Baker, B., & Shostak, R. (1972). Gossips and telephones. *Discrete Mathematics*, *2*(3), 191–193. https://doi.org/10/bddz44

van den Berg, L., & Gattinger, M. (2020). Dealing with unreliable agents in dynamic gossip. In M. A. Martins & I. Sedlár (Eds.), *Dynamic logic. new trends and applications* (pp. 51–67). Springer International Publishing. https://doi.org/fp2t

van den Berg, L. (2020). *Unreliable gossip*. Retrieved January 24, 2021, from https://github.com/linevdberg/UnreliableGossip

Cisco. (2020, March). *Cisco annual internet report (2018–2023)* (White Paper No. C11-741490-01). Retrieved January 4, 2020, from https://www.cisco.com/c/en/us/solutions/collateral/executive-perspectives/annual-internet-report/white-paper-c11-741490.pdf

Cooper, M. C., Herzig, A., Maris, F., & Vianey, J. (2019). Temporal epistemic gossip problems. In M. Slavkovik (Ed.), *Multi-agent systems* (pp. 1–14). Springer International Publishing. https://doi.org/fp2v

Czaplicki, E., & Chong, S. (2013). Asynchronous functional reactive programming for GUIs. *SIGPLAN Not.*, *48*(6), 411–422. https://doi.org/10/f45mkb

Das, A., Gupta, I., & Motivala, A. (2002). SWIM: Scalable weakly-consistent infection-style process group membership protocol. *Proceedings International Conference on Dependable Systems and Networks*, 303–312. https://doi.org/10/dfvrrz

DeCandia, G., Hastorun, D., Jampani, M., Kakulapati, G., Lakshman, A., Pilchin, A., Sivasubramanian, S., Vosshall, P., & Vogels, W. (2007). Dynamo: Amazon's highly available key-value store. *Proceedings of twenty-first ACM SIGOPS symposium on Operating systems principles*, 205–220. https://doi.org/10/bj8wqc

Demers, A., Greene, D., Houser, C., Irish, W., Larson, J., Shenker, S., Sturgis, H., Swinehart, D., & Terry, D. (1988). Epidemic algorithms for replicated database maintenance. *ACM SIGOPS Operating Systems Review*, *22*(1), 8–32. https://doi.org/10/fmhpvg

van Ditmarsch, H., van Eijck, J., Pardo, P., Ramezanian, R., & Schwarzentruber, F. (2018). Dynamic gossip. *Bulletin of the Iranian Mathematical Society*, *45*(3), 701–728. https://doi.org/10/cvpm

van Ditmarsch, H., Gattinger, M., Kuijer, L. B., & Pardo, P. (2019). Strengthening gossip protocols using protocol-dependent knowledge. *Journal of Applied Logics*, *6*(1), 157–203. https://arxiv.org/abs/1907.12321

Gattinger, M. (2016, July). *webGossip*. Retrieved December 24, 2020, from https://w4eg.de/malvin/illc/webgossip

Gattinger, M. (2017). *Explicit epistemic model checking for dynamic gossip*. Retrieved December 24, 2020, from https://github.com/m4lvin/GoMoChe

Graham, R. L. (1973). Review of the article "On a telephone problem" by R. Tijdeman. *Mathematical Reviews*, *0342405*.

Haas, Z. J., Halpern, J. Y., & Li Li. (2006). Gossip-based ad hoc routing. *IEEE/ACM Transactions on Networking*, *14*(3), 479–491. https://doi.org/10/b269rq

Hajnal, A., Milner, E. C., & Szemerédi, E. (1972). A cure for the telephone disease. *Canadian Mathematical Bulletin*, *15*(3), 447–450. https://doi.org/10/cpr4cv

Harary, F., & Schwenk, A. J. (1974). The communication problem on graphs and digraphs. *Journal of the Franklin Institute*, *297*(6), 491–495. https://doi.org/10/djhtxr

Herzig, A., & Maffre, F. (2017). How to share knowledge by gossiping. *AI Communications*, *30*(1), 1–17. https://doi.org/10/f94qxh

Kermack, W. O., & McKendrick, A. G. (1927). A contribution to the mathematical theory of epidemics. *Proceedings of the Royal Society of London*, *115*(772), 700–721. https://doi.org/10/fwx2qw

Kermarrec, A.-M., & van Steen, M. (2007). Gossiping in distributed systems. *ACM SIGOPS Operating Systems Review*, *41*(5), 2–7. https://doi.org/10/d3zffh

Lebensold, K. (1973). Efficient communication by phone calls. *Studies in Applied Mathematics*, *52*(4), 345–358. https://doi.org/10/ghrv4s

Liben-Nowell, D. (2002). Gossip is synteny: Incomplete gossip and the syntenic distance between genomes. *Journal of Algorithms*, *43*(2), 264–283. https://doi.org/10/bkq8p9

Maffre, F. (2016). *Gossip problem PDDL-generator*. Retrieved December 24, 2020, from https://github.com/FaustineMaffre/GossipProblem-PDDL-generator

Moelker, R. (2016). *Visualization of gossip protocol simulation*. Retrieved December 24, 2020, from https://github.com/RRMoelker/gossip-visualization

Renesse, R. v. (2016). A blockchain based on gossip? – a position paper. *Distributed Cryptocurrencies and Consensus Ledgers (DCCL 2016)*. https://www.zurich.ibm.com/dccl/papers/renesse%5C%5Fdccl.pdf

Tijdeman, R. (1971). On a telephone problem. *Nieuw Archief voor Wiskunde*, *3*(19), 188–192.

A Survey questions

1. I am familiar with dynamic and/or distributed gossip.

   Sem. Diff. Scale: 1 = Not familiar at all , 5 = Very familiar

2. What do you think of the tool's visual design?

   Sem. Diff. Scale: 1 = Very bad, 5 = Very good

3. How much time does it take you to understand what each part of the interface is for?

   Sem. Diff. Scale: 1 = Very little time, 5 = Very much time

4. Do you think the error messages shown when providing incorrect input are helpful?

   Sem. Diff. Scale: 1 = Not helpful at all, 5 = Very helpful

5. Is there any part of the interface you think could use improvement? If so, please briefly explain which part and why.

   Text input field

6. Do you think this tool could be useful in a research setting?

   Yes/No/Maybe

7. Do you think this tool could be useful in a study setting?

   Yes/No/Maybe

8. Which features do you think are the most important in this tool?

   List of options:

   - *Visualising gossip graphs;*
   - *Checking permissibility of call sequences on gossip graphs under certain protocols;*
   - *Executing call sequences on gossip graphs;*
   - *Navigating between different states of a gossip graphs after (a) call(s) have been made;*
   - *Gossip graph statistics: whether a graph is a sun graph, and whether a relation is strongly/weakly connected;*
   - **Creating custom protocols by creating a boolean combination of constituents (as defined in the paper "Dynamic Gossip" by van Ditmarsch et al. (2019));*
   - **Generating execution trees for gossip protocols;*
   - **Exporting gossip graphs to LaTeX or GraphViz formats;*
   - **Exporting execution tree diagrams to LaTeX or GraphViz formats;*
   - *Other (Text input field).*

   The asterisk (*) indicates functionality that was not finished at the time the survey was sent out.

9. Are there any features that you would like to see added or changed?

   Text input field