Sprinkler: A probabilistic dissemination protocol to provide fluid user interaction in multi-device ecosystems

Adrien Luxey, Yérom-David Bromberg, Fábio M Costa, Vinícius Lima, Ricardo da Rocha, François Taïani

Sprinkler: A probabilistic dissemination protocol to provide fluid user interaction in multi-device ecosystems

Adrien Luxey*, Yérom-David Bromberg*, Fábio M. Costa†, Vinícius Lima†, Ricardo C.A. da Rocha†, François Taïani*
*Univ. of Rennes 1 / IRISA / Inria, Rennes, France, †Univ. Federal de Goiás, Goiânia-GO, Brazil
*{firstname.surname}@irisa.fr, †fmc@inf.ufg.br, viniciusb50,@gmail.com, rcarocha@ufg.br

***Abstract*—Offering fluid multi-device interactions to users while protecting their privacy largely remains an ongoing challenge. Existing approaches typically use a peer-to-peer design and flood session information over the network, resulting in costly and often unpractical solutions. In this paper, we propose SPRINKLER, a decentralized probabilistic dissemination protocol that uses a gossip-based learning algorithm to intelligently propagate session information to devices a user is most likely to use next. Our solution allows designers to efficiently trade off network costs for fluidity, and is for instance able to reduce network costs by up to 80% against a flooding strategy while maintaining a fluid user experience.**

I. INTRODUCTION

These last decades have witnessed an exponential proliferation of devices connected to the Internet. We are entering a new era, a further step towards Mark Weiser's vision of ubiquitous computing [39], where the paradigm of a single user for a single device no longer applies. Users own multiple devices (PCs, smartphones, smart watches, tablets, notebooks), and no longer spend their time on a single desktop screen to perform their daily digital activities (browsing, searching, online shopping and gaming, video streaming) [20], [25].

This emerging trend is far from being without challenges. Accessing everything, anytime, anywhere, in a continuous manner across various heterogeneous devices at the right time and at the right place, may become a daunting task for users. Particularly, support for streamlining the user experience is lagging behind. As introduced by Levin [33], applications targeting a multi-device ecosystem should be designed with three key fundamental concepts in mind: *Consistency*, *Continuity*, and *Complementarity* (for short the *3Cs*). For instance, one application may not have a *consistent* user interface across heterogeneous devices, undermining the user's multi-device interaction, and hence increasing user frustration. Similarly, with applications not designed for *continuity*, users may have to manage themselves how to resume their interaction context from their current device to the next one. This implies, most often, that users have to manually transfer up-to-date application data, back and forth across devices, in order to recreate interaction sessions, thus compromising the fluidity of interaction. Finally, multiple devices may be used simultaneously to *complement* each other, e.g., one device may act as a remote control to pilot another device.

This paper explicitly targets the issue of providing *continuous* interaction. Indeed, with the latest advances in Web technologies, we consider that the problem of designing *consistent* applications is almost solved. In particular, through the use of HTML, CSS, and JAVASCRIPT natively embedded in Web browsers, web-based applications have become mostly consistent across heterogeneous devices [2], [4], [8], and are now the most adequate candidates to support users' multi-device experience [11], [13], [21], [22], [30], [34], [37].

Additionally, from a recent study led by Google [20], 90% of users who grew up in a multi-device ecosystem, switch between devices in a sequential way. In other terms, a given application may only be active on a single device at a given point in time. The prevalence of sequential usage leads us: (i) to focus our work on sequential interactions, and (ii) to consider the simultaneous usage of devices, i.e the *complementary* aspect of users' interaction, for future works.

Finally, the problem of providing continuous interaction across heterogeneous devices is currently massively addressed by relying *de facto* on cloud providers that act as a central point of synchronization. For instance, popular applications such as EVERNOTE [6], 1PASSWORD [1], WUNDERLIST [9], CHROME [7], AMAZON KINDLE [3], etc. rely on either GOOGLE DRIVE, DROPBOX, ICLOUD, AMAZON S3 and/or their own servers to perform *session handoff*, i.e to save and transfer the interactive session of an application in the multi-device ecosystem. However, such solutions suffer from a key shortcoming: users' activity and their related personal and private information are recorded by application providers. Users do not expect to have their digital life tracked and analyzed by a third party that collects data on their behalf.

To the best of our knowledge, only few approaches have sought to protect users' privacy in a multi-device ecosystem. These approaches typically exploit a peer-to-peer architecture to save and transfer application sessions only on devices owned by users, avoiding the need to trust any third party [17], [18], [32]. However, as the behavior of users is not known in advance, the aforementioned solutions blindly flood the ongoing interactive session to all devices of the ecosystem, as no device is able to predict which device will be used next. Such a brute-force broadcast mechanism, despite its simplicity, leads to very poor performances as it implies: (i) redundant

messages, (ii) overconsumption of network bandwidth, (iii) a higher latency that inherently impacts the fluidity of user interaction, and (iv) faster energy depletion. Further, it does not scale well with the number of devices. Although the current number of devices owned by a user is on average around 4 devices [20], this number is expected to increase with the advent of the Internet Of Things.

In this paper, we introduce SPRINKLER, a novel approach to perform predictive session handoff by learning in a *decentralized* manner how the user behaves. SPRINKLER combines a probabilistic dissemination protocol and a proactive session handoff mechanism in order to provide seamlessly fluid user interactions in a multi-device ecosystem. Our solution: (i) does not rely on any centralization point, (ii) has a bounded and known network resource consumption, (iii) is able to predict which device is the most likely to be used next, enabling the transfer of the ongoing interaction session without blind flooding, (iv) respects the user's right to privacy, and (v) scales to an arbitrary number of devices.

Our contributions are as follows:

- We have designed SPRINKLER, a protocol based on two algorithms: the SPRINKLER *Gossiper*, which allows devices to gain knowledge of the user's behavior in a distributed manner; and the SPRINKLER *Session Handoff* mechanism, which uses this knowledge to proactively send chunks of the current session to devices that will most probably be used next;
- We have evaluated our approach with 8 different discrete time Markov models to emulate user behaviors;
- We have demonstrated that there is no ideal dissemination protocol. It is all about the tradeoff between prediction accuracy, latency, and network consumption. We show in particular that the performance of SPRINKLER greatly depends on the user's behavior. The more predictable a user is, the better SPRINKLER performs;
- SPRINKLER allows designers to efficiently trade off network costs for fluidity, and is for instance able to reduce network costs by up to 80% against a flooding strategy while maintaining a fluid user experience.

In the following we detail SPRINKLER's concepts and approach (Section II). Section III then evaluates the proposed approach regarding performance and different models emulating different types of user behavior. Finally, Section IV considers related work, and Section V discusses future work and conclusions.

II. CONCEPTS AND SOLUTION APPROACH

Our goal is to provide a communication protocol such that every time the user (that we call Alice) opens one of her devices, she finds her Web applications as she left them, regardless of the device she has used previously. Because the day-to-day usage information of one's appliances is a private asset, we wish to avoid relying on an external storage system, which would threaten the user's right to privacy. To this end, our protocol should only involve Alice's devices, by leveraging distributed communication strategies. Secondly, we want our protocol to be lightweight, to avoid draining power from the mobile appliances running it.

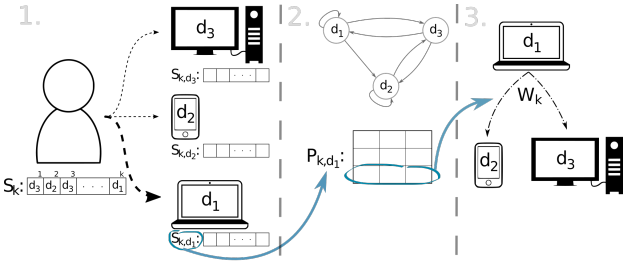

Fig. 1. Pipeline of the Session Handoff: from user behavior to model inference to sharing of the session data.

The state of the user's applications is stored in a blob, arbitrarily heavy in size, thereafter called a *session*. Our protocol, SPRINKLER, *proactively* shares portions (*chunks*) of the user's session whenever she leaves a device, by *predicting* the device she will use next. This prediction is achieved by letting devices *learn* the user's behavior by gossiping information among themselves. We wish to minimize, on one hand, the time Alice has to wait for her previous session to be fetched when she opens a new device, and on the other hand, the network traffic induced by session exchanges.

Towards this aim, our protocol is constituted of two algorithms: the SPRINKLER *Gossiper* will allow devices to gain knowledge of the user's behavior; the SPRINKLER *Session Handoff* algorithm will use this information to *proactively* send chunks of the current session to devices that will most probably be used next.

A. Overall approach

We consider a user, Alice, who uses a number of N devices, $\mathcal{D} = \{d_1, ..., d_N\}$. We can model Alice's use of her devices as a sequence of *interactions*: $S = \{r_1, r_2, ..., r_i, ...\}$. Each interaction r_i is characterized by a pair $(d_{r_i}, t_{r_i}) \in \mathcal{D} \times \mathbb{R}$, which means that Alice started using the device d_{r_i} at time t_{r_i} (and stopped using it before $t_{r_{i+1}}$). We assume that Alice only uses one device at a time, such that no two interactions share the same timestamp.

For the purpose of our experiments, we consider that Alice's devices have access to a synchronized physical clock, creating a total order on the sequence. The same total order could be obtained with logical clocks, e.g. Lamport timestamps [29], since interactions are never concurrent.

Our Session Handoff procedure is illustrated in Fig. 1.

Propagating the user's behavior: From a global point of view, the sequence S_k contains all the k interactions performed by the user since the beginning of the program's execution. Locally, however, each device d initially only knows about the sequence $S_k|_d$ of interactions that took place on it, that is:

$$S_k|_d = \{r \in S_k,\ r{\downarrow}\mathcal{D} = d\} \implies S_k = \bigcup_{d \in \mathcal{D}} S_k|_d,$$

where $r{\downarrow}\mathcal{D}$ represents the projection of interaction r on the set of devices, i.e., the device on which r took place.

To gain knowledge on the user's behavior, the devices gossip information about interactions among themselves. This way, at step k, every device d knows a (possibly incomplete) local sequence $S_{k,d}$ of the user's actions, such that:

$$S_k|_d \subseteq S_{k,d} \subseteq S_k.$$

Inferring a probabilistic model: An ordered sequence S of interactions can be used to compute a discrete time Markov chain, representing the probability that the user swaps from a device to another, for each pair of devices in $\mathcal{D}$. To do so, we firstly need to compute the matrix of transition counts $C = (c_{d_i,d_j})_{(d_i,d_j)\in\mathcal{D}^2}$ between devices:

$$c_{d_i,d_j} = |\{r_t, r_{t+1} \in S,\ (r_t{\downarrow}\mathcal{D}) = d_i \wedge (r_{t+1}{\downarrow}\mathcal{D}) = d_j\}|\,.$$

C captures the number of times Alice switches between each pair of devices (d_i, d_j) in S. From C, we can derive the matrix of transition probabilities $P = (p_{d_i,d_j})_{(d_i,d_j)\in\mathcal{D}^2}$, (i.e. the weights of the Markov chain's edges):

$$p_{d_i,d_j} = \frac{c_{d_i,d_j}}{\sum_{d_k\in\mathcal{D}} c_{d_i,d_k}} = \mathrm{P}\left[d_i \to d_j\right],$$

where $d_i \to d_j$ means "Alice uses the device d_j right after d_i". p_{d_i,d_j} thus represents the probability that Alice switches from d_i to d_j, according to the sequence of the user's interactions, S. We call Ψ the operation of generating a Markov transition matrix P from a sequence S: $P = \Psi(S)$.

A device d_{curr} that is currently being used by Alice at step k (i.e. d_1 in Fig. 1) uses its local sequence $S_{k,d_{\text{curr}}}$ to compute the transition matrix $P_{k,d_{\text{curr}}} = \Psi\left(S_{k,d_{\text{curr}}}\right)$. This provides d_{curr} with the transition vector $\mathbf{p}_k$ that contains, for each $d \in \mathcal{D}$, the probability that the user will switch from d_{curr} to d:

$$\mathbf{p}_k = P_{k,d_{\text{curr}}}\left(d_{\text{curr}}, *\right) = \left(\mathrm{P}\left[d_{\text{curr}} \to d \mid S_{k,d_{\text{curr}}}\right]\right)_{d\in\mathcal{D}}. \quad (1)$$

Note that $\mathbf{p}_k\left(d_{\text{curr}}\right)$ is usually not null: the user sometimes switches back to the same device.

Performing Session Handoff: After Alice closes her current device d_{curr}, we want to proactively send the blob containing her application state—her *session*—to the next device she will use: d_{next}.

Throughout the rest of the article, we assume that every session weighs w_{sess} bytes. Because d_{curr} cannot be sure of which device will be used next, it sends portions of its session to several selected peers. We call these portions *session chunks*, and assume that they can weigh any size from 0 to w_{sess}. Finally, we define the set $\mathcal{D}'$ of devices to which d_{curr} can potentially send session chunks: $\mathcal{D}' = \mathcal{D} \setminus \{d_{\text{curr}}\}$.

d_{curr} sends $w_{k,d} \in [0, w_{\text{sess}}]$ bytes of the session to each device $d \in \mathcal{D}'$, resulting in the vector W_k of all data chunks sent by d_{curr} at step k:

$$W_k = \left(w_{k,d} \in [0, w_{\text{sess}}]\right)_{d\in\mathcal{D}'} = f\left(\mathbf{p}_k\right). \quad (2)$$

The performance of the Session Handoff depends on the accuracy of $\mathbf{p}_k$, which depends on the local sequence of d_{curr}, $S_{k,d_{\text{curr}}}$. In the following, we first present SPRINKLER Gossiper, which reliably propagates a user's sequence of interactions to all devices, before discussing SPRINKLER Session Handoff, which handles the proactive migration of a user's session.

B. Decentralized knowledge aggregation

Initially, a device can only observe interactions taking place locally. In order to predict a user's future behavior, all devices must, however, gain a global overview of the user's past behavior, and thus aggregate their local knowledge into a global interaction sequence. We propose to perform this aggregation with a probabilistic dissemination protocol [12], [15], [26] that we have called SPRINKLER Gossiper.

1) Intuition: The Gossiper implements a *reactive* and *incremental* aggregation that involves all of a user's devices. The protocol is invoked every time the user (say Alice) leaves a device to move to another one, signaling the end of an interaction, and the start of a new one. SPRINKLER Gossiper is *gossip-based*, i.e., it uses randomized message exchanges between devices to propagate the sequence of interactions performed by Alice. This randomized approach makes our protocol both *lightweight* and *robust*, two properties that are central to decentralized session handoff. We use a *push-pull* strategy [23] to propagate information, i.e., when a device p contacts a device q, p sends new information to q (*push*), but also requests any new information q might have (*pull*).

In order to avoid redundant communication rounds, the Gossiper further keeps track of each device's local perception of other device's knowledge using a mechanism inspired from vector clocks [31] combined with incremental diffs.

2) Algorithm: The pseudo-code of SPRINKLER Gossiper is shown in Figures 2 and 3. To ease our explanation, the *request* part of the push/pull exchange is shown in Fig. 2 from the point of view of p, while the *reply* part is shown in Fig. 3 from the point of view of q. (All devices execute both parts in practice.) We assume that p and q are owned by Alice, and that she is currently using device p. p's current knowledge of Alice's sequence of interactions is stored in variable S_p, while the array $RV_p[\cdot]$ stores p's remote view of other devices' knowledge of Alice's sequence (Table I).

The algorithm starts when Alice begins a new interaction by opening device p (line 1). The algorithm inserts a new interaction record $\langle p, \text{timestamp}\rangle$ into p's local interaction view S_p (lines 3-4), and launches the probabilistic dissemination, implemented in GOSSIPUPDATE().

GOSSIPUPDATE() first selects a small random set of f other devices from p's local sequence S_p (lines 6-8). As a consequence, the only devices that participate in SPRINKLER are the ones that Alice already used at least once.

These random devices are selected from S_p, i.e., the sequence of interactions already learned by p, from which we exclude p and the most recent device found in S_p (which is likely to be up to date). p initiates a push/pull exchange with each selected peer q which is not known to know at least as much as p (lines 9-13). This *knowledge check* is performed at lines 10-11, using $RV_p[q]$, p's idea of the interactions that are known to q. By construction, and in the

TABLE I
VARIABLES & PARAMETERS OF SPRINKLER

	Variables maintained by the device p belonging to Alice
S_p	p's knowledge of Alice's interaction sequence, i.e. its *local sequence*. S_p is initialized with a small number of core devices (possibly only one) that Alice uses regularly.
$RV_p[\cdot]$	$RV_p[q]$ contains p's idea of what is known to device q. Initially $RV_p[q] = \varnothing$ for all $q \in \mathcal{D} \setminus \{p\}$.
	Parameters of the algorithm
f	The *fanout* of the probabilistic broadcast, that is the number of devices that each device communicates new information with.

```
1:  on event Alice opens device p
2:      r ← ⟨p, timestamp⟩                          ▷ New interaction r
3:      S_p ← S_p ∪ {r}                             ▷ Updating p's local view
4:      GOSSIPUPDATE()                              ▷ Triggering the dissemination

5:  function GOSSIPUPDATE(exclude[= ∅])
6:      last_device ← most recent device in S_p
7:      exclude ← exclude ∪ {p, last_device}
8:      peers ← f devices from {devices from S_p} \ exclude
9:      for q ∈ peers do                            ▷ Looping through random peers
10:         S_diffpush ← S_p \ RV_p[q]
11:         if |S_diffpush| > 0 then                ▷ Only sending new data
12:             send ⟨REQ : S_diffpush⟩ to q        ▷ Push/pull to q
13:             RV_p[q] ← RV_p[q] ∪ S_diffpush      ▷ Tracking q

14: on receive ⟨ANS : S_diffpull⟩ from q:           ▷ Pull reply
15:     S_p ← S_p ∪ S_diffpull
16:     RV_p[q] ← RV_p[q] ∪ S_diffpull
```

Fig. 2. SPRINKLER's push/pull request (on device p)

```
17: on receive ⟨REQ : S_diffpush⟩ from p:           ▷ Push/pull request
18:     RV_q[p] ← RV_q[p] ∪ S_diffpush              ▷ Tracking p
19:     if S_diffpush ⊈ S_q then                    ▷ Is S_diffpush new?
20:         S_q ← S_q ∪ S_diffpush
21:         GOSSIPUPDATE({p})                       ▷ Propagating new data
22:     S_diffpull ← S_q \ RV_q[p]
23:     if |S_diffpull| > 0 then                    ▷ Anything new for p?
24:         send ⟨ANS : S_diffpull⟩ to p            ▷ Answering pull
25:         RV_q[p] ← RV_q[p] ∪ S_diffpull
```

Fig. 3. SPRINKLER's push/pull reply (on device q)

absence of communication faults, $RV_p[q]$ underestimates q's actual knowledge (i.e., $RV_p[q] \subseteq S_q$)[1], which means no new information is missed.

The **send** operation at line 12 starts the actual push/pull exchange with q: the incremental update S_{diffpush} is sent to q. On receiving S_{diffpush} (line 17, Fig. 3), q first processes p's incremental update (lines 18-21), by *(i)* adding it to q's idea of p's view (line 18), *(ii)* updating its own view if needed (line 20), and *(iii)* launching a cascading dissemination in case the diff contains information new to q. The condition at line 19

[1]This is because any interactions added to $RV_p[q]$ by p have either been sent to q (lines 13 and 25) or received from q (line 16).

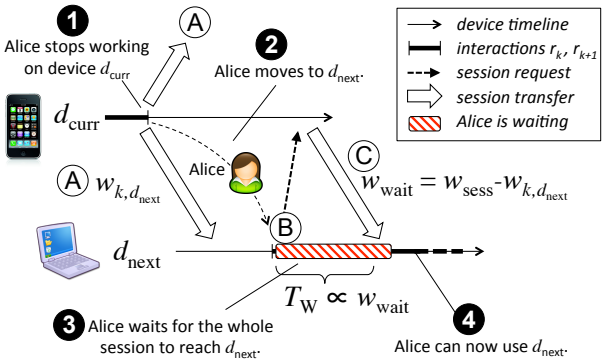

Fig. 4. Timeline of the session handoff from device d_{curr} to d_{next}.

ensures the dissemination eventually stops, as increments are never gossiped twice by the same device. $\{p\}$ is passed as a parameter to GOSSIPUPDATE() to increase the probability of hitting uninformed devices.

Lines 22-25 implement q's reply to p's pull request. Again, q only replies to p if it might possess new information (test of line 23), in order to reduce communication. The (possible) reply from q to p is processed by p at lines 14-16 (Fig. 2).

3) Bootstrap and reliability of SPRINKLER *Gossiper:* Because a device p only gossips with other devices found in its sequence view S_p, p's view needs to be initialized to a default value of a few core devices regularly used by Alice e.g. $\{\langle a, 0\rangle, \langle b, 0\rangle\}$, where 0 is an arbitrary bootstrapping timestamp. The use of S_p as a source of gossiping candidates prevents devices not used by Alice to be involved in the protocol. When a new device is used by Alice, on the other hand, it propagates updates containing at least itself (lines 2-3), and automatically becomes known to the rest of the system.

The overall reliability of the gossip diffusion is governed by its fanout coefficient f (which appears at line 8). In a reliable network, a fanout f slightly over $\log(N)$ ensures that all devices will be reached with a very high probability [27]. In practice we therefore use $f = \lceil \log(N) \rceil$.

Although we assume a reliable network, transient communication failures might occur. When this happens, the protocol might temporarily fail to reach all nodes, but full propagation will resume when the network recovers. While the network is degraded, $RV_p[q]$ might diverge, and might contain information not included in S_q. This is because we do not insure that the message $\langle \text{REQ} : S_{\text{diffpush}} \rangle$ sent at line 12 is successfully received by q before modifying p's remote view $RV_p[q]$. In most situations, however, other nodes will provide q with the missed information when the network recovers. This choice favors communication lightness over reliability, but turns out to work well in practice (as shown in Section III-B1).

C. The Session Handoff algorithm

Fig. 4 shows the different steps undertaken by SPRINKLER Session Handoff when Alice moves from device d_{curr} (her mobile phone here) to another device d_{next} (her laptop) between interactions r_k (on device d_{curr}) and r_{k+1} (on device d_{next}).

When Alice leaves her mobile phone (d_{curr}, label ❶) at the end of interaction r_k, SPRINKLER proactively sends a *partial session state* to her other devices (label Ⓐ). (We assume here that we can detect the end of an interaction, e.g., using some activity recognition mechanism.) The amount of state each device receives is decided using the user behavioral model constructed so far by SPRINKLER Gossiper. (We detail this below.) We note $w_{k,d}$ the amount of session state proactively received by device d at the end of interaction r_k. In Fig. 4, Alice's laptop proactively receives $w_{k,d_{\text{next}}}$ of Alice's session on her mobile phone.

When Alice reaches her laptop ❷, SPRINKLER Session Handoff reactively requests the remainder of the session state that has not reached the laptop yet Ⓑ[2]. While the remaining session state $w_{\text{wait}} = w_{\text{sess}} - w_{k,d_{\text{next}}}$ is downloaded from d_{curr}, Alice must wait (❸, hashed bar on the laptop timeline) until she can finally use her device ❹. Assuming latency is negligible compared to the download time of w_{wait}, the waiting time of Alice T_W is proportional to w_{wait}.

The perfect session handoff algorithm would always provide its user with the last state of her applications when she opens a device, whichever of her appliances she has previously used, and without any waiting time. In addition, given that at least some of her devices are mobile assets with limited resources, this algorithm should consume no more than the bare minimum: it would have sent the application session once, from the device Alice just quit to the one she is about to use.

Such an algorithm is not feasible, since no one can predict the future. Instead, the current device d_{curr} infers which devices are most likely to be used next, to *proactively* send them chunks of the session's blob when the user quits d_{curr}. If d_{curr} sends more session chunks to the other devices, the waiting time T_W will lower at the cost of higher network traffic. On the other hand, if d_{curr} does not send any of the current's session state, the waiting cost will be maximal, as d_{next} will have to reactively fetch the entire session when she opens it. We call this *network cost* C_N. There is clearly a trade-off between the waiting time T_W and the network traffic C_N.

1) Formulating the handoff cost: Thanks to the Gossiper, d_{curr} knows a subset of the user sequence of interactions $S_{k,d_{\text{curr}}}$. By computing the transition vector $\mathbf{p}_k$ (see Equation 1) out of $S_{k,d_{\text{curr}}}$, the Session Handoff algorithm outputs the data vector W_k (see Equation 2), that contains the amount of bytes of the session to send to each other device.

We formulate the waiting time T_W and the network cost C_N described earlier:

- $T_W \propto w_{\text{wait}} = w_{\text{sess}} - w_{k,d_{\text{next}}}$: The waiting time T_W is proportional to the quantity of the current session that the next device d_{next} still needs to download. In the particular case where $d_{\text{curr}} = d_{\text{next}}$, the session is already fully on the device, leading to a null waiting cost $T_W = 0$;
- $C_N = \sum_{d \in \mathcal{D}'} w_{k,d}$: the network cost C_N is the sum of all the session chunks proactively sent from d_{curr} to the other appliances in $\mathcal{D}'$.

2) Computing W_k: The goal of the Session Handoff algorithm is to figure out the best vector of sent data W_k based on the transition vector $\mathbf{p}_k$. d_{curr} wants to minimize T_W given that it will send a total of C_N bytes of the session to its peers. We introduce the parameter γ, that controls the amount of data the device d_{curr} will send to the other appliances.

We propose two different solutions for the calculus of W_k:

- **Uniform**: As a baseline, our first solution is to send the same quantity of the session to each device in $\mathcal{D}'$, regardless of $\mathbf{p}_k$:

$$\forall d \in \mathcal{D}', w_{k,d} = w_{\text{sess}} * \frac{\gamma}{N-1}.$$

- **Proportional**: Our second solution is to make W_k proportional to $\mathbf{p}_k$. This way, devices having a high probability of being used next will naturally receive a bigger portion of the user's session. The most simple computation would be the following:

$$\forall d \in \mathcal{D}', w_{k,d} = w_{\text{sess}} * \min(\gamma * \mathbf{p}_k(d), 1).$$

However, a device d_{low} may have a very low probability $\mathbf{p}_k(d_{\text{low}})$ of being chosen. The preceding calculus would result in a negligible $w_{k,d_{\text{low}}}$ compared to the cost of the network exchange. Therefore, we compute a new transition vector $\mathbf{p'}_k$, such that any probability inferior to a certain threshold α is null. In practice, we arbitrarily set $\alpha = 1/(N-1)$. We introduce a second parameter β to ensure that $\mathbf{p'}_k$ sums to one. It is computed as follows:

$$\beta = \frac{1}{\sum_{\substack{d \in \mathcal{D}' \\ \mathbf{p}_k(d) > \alpha}} \mathbf{p}_k(d)};$$

$$\forall d \in \mathcal{D}', \quad \mathbf{p'}_k(d) = \begin{cases} 0 & \text{if } \mathbf{p}_k(d) \leq \alpha \\ \beta * \mathbf{p}_k(d) & \text{else} \end{cases}.$$

The vector of sent data is then simply proportional to this new pruned vector of probabilities:

$$\forall d \in \mathcal{D}', w_{k,d} = w_{\text{sess}} * \min(\gamma * \mathbf{p'}_k(d), 1).$$

This way, we send session chunks only to devices that have a high enough probability of being used.

We have proposed two solutions to dispatch session chunks to the devices on session handoff. The first (uniform) will give us a comparison point to observe the influence of the behavioral knowledge inferred with SPRINKLER Gossiper (in the proportional approach). The parameter γ will allow us to tune the overall quantity of data that d_{curr} will send to its

[2]In order to fetch this remaining state, d_{next} must know the address of d_{curr}. It can be achieved by several means: if d_{curr} sent chunks of the last session to d_{next} (which is not the case when $w_{k,d_{\text{next}}} = 0$) or by looking at the penultimate interaction in $S_{k+1,d_{\text{next}}}$ (though d_{next}'s local sequence can be incomplete). In addition, when d_{next} wrongly believes that a given device d is the device used at interaction r_k and asks it the last session, d uses its own knowledge to redirect d_{next} to the right device, d_{curr}. If d_{next} was still unable to locate d_{curr}, the Session Handoff would fail completely: Alice's previous session would never be loaded on d_{next}. Consequently, we evaluate a reactive handoff in Section III-B3.

peers: from $\gamma = 0$, that solely relies on reactive handoff, to a maximal γ, that consists of sending the entirety of the session to each device having a non-null probability of being used next.

III. Evaluation

A. Experimental approach

We evaluate our contributions by launching several virtual devices used by an emulated user. In our futuristic scenario, we imagine a user controlling a dozen of devices sequentially. In the following, we first propose several behavioral models that emulate a fictitious user's activity, before presenting our experimental setup.

1) Proposed user behavior models: In Section II-A, we have represented a user's behavior as a growing sequence S of interactions with her appliances. To emulate these interactions, we propose to use a number of discrete-time Markov models $\mathcal{M}$, from which we then generate the sequences of interactions driving the evaluation of our contributions. Given the set $\mathcal{D} = \{d_1, \ldots, d_N\}$ of a user's devices, a discrete-time Markov user model $\mathcal{P}_\mathcal{M}$ takes the form:

$$\mathcal{P}_\mathcal{M} = \left(p_{d_i,d_j}\right)_{(d_i,d_j)\in\mathcal{D}^2} \quad \text{s.t.} \quad p_{d_i,d_j} = \mathrm{P}\left[d_i \rightarrow d_j\right].$$

We propose to use 5 different strategies to generate these user models, and for 3 of these, we create two variants, leading to a total of 8 user models. These models differ in terms of *density* (representing how many potential next devices might be picked after the current one) and *uniformity* (representing the extent to which selection of a next device is biased or not).

Figure 5 shows examples of the transition matrices $\mathcal{P}_\mathcal{M}$ generated by the 8 models, represented as heatmaps (using $N = 12$ devices). The creation strategies are explicated below:

1) **uniform:** The worst case scenario for our framework is a completely uniform model, where Alice chooses her next device with an even probability of $1/N$. In this situation, the currently used device cannot guess what appliance will be used next, making the session handoff as good as random;
2) **cyclic:** The best usage pattern is when Alice uses her devices in a cyclic order (making a circular Markov chain), because the devices always succeed in the prediction of the appliance she will use next. In this model, every transition vector $\mathcal{P}_\mathcal{M}(d, *)$ is constant;
3) **sequence:** This model is computed from a random model sequence $S_\mathcal{M}$ containing l interactions. $S_\mathcal{M}$ is populated by devices randomly selected with an uneven probability $P_{\text{devices}} \in [0,1]^N$. P_{devices} thus favors the use of certain devices (e.g. Alice uses her smart-phone more often than her mother's laptop). We compute the transition matrix $\mathcal{P}_\mathcal{M} = \Psi(S_\mathcal{M})$ as was shown in Section II-A. The longer the model sequence $S_\mathcal{M}$, the denser the output matrix. We thus generated two *sequence* models: **3.1**, with $l = 2 * N$, and **3.2**, with $l = 10 * N$;
4) **zipf:** Many processes in real life follow a Zipf law [36]: word occurrences, citations of scientific articles, wealth per inhabitant... Zipf's discrete law of probability $z(k; n, s)$ is defined by a constant population size $n \in \mathbb{N}$, a rank $k \in \mathbb{N}^+$, and a constant exponent $s \in [1, +\infty[$. It states that: $z(k; n, s) = z(1; n, s) * k^{-s}$, i.e., the probability of occurrence of the k^{th} most frequent element $z(k; n, s)$ equals the probability of occurrence of the most frequent element $z(1; n, s)$ times k^{-s}. The bigger the exponent s, the faster the function approaches zero, and the more $z(1; n, s)$ dominates the other probabilities. In our futuristic scenario where a user would own and frequently use a dozen of devices, the assumption that the transition probability between her devices follows a Zipf law seems plausible.
We propose a model where, to each device d's transition vector $\mathcal{P}_\mathcal{M}(d, *)$, we assign a random permutation of Zipf's law's PMF using $n = N$ and a random exponent s. We propose two variants: in model **4.1**, we pick s from $[1, 2.5]$, which keeps the biggest probability $z(1; n, s) \in [0.32, 0.75]$. In model **4.2**, we draw s from $[2.5, 5]$, such that $z(1; n, s) \in [0.75, 0.96]$. Note that $\mathcal{P}_\mathcal{M}$ is always dense using the zipf model, but the probabilities' heterogeneity grows with the exponent s;
5) **sparse:** A sparse transition matrix contains null probabilities: there are certain devices d_i and d_j such that d_j will never be used after d_i. This is realistic, e.g., two desktop computers from two faraway locations will never be accessed in a row. For SPRINKLER's handoff, this sparsity prevents the used device to have to choose between too many appliances to send session chunks to. For each $d \in \mathcal{D}$, we compute the transition vector $\mathcal{P}_\mathcal{M}(d, *)$ by drawing samples from a Zipf law $Z(n, s)$ with a "big" n (e.g. 1000) and a fixed s:

$$\mathcal{P}_\mathcal{M}(d, *) = \frac{X}{\sum\limits_{x\in X} x} \text{ s.t. } \begin{cases} X = \{Z(n, s) - 1\}^N \\ \sum\limits_{x\in X} x \neq 0 \end{cases}$$

The bigger the exponent s, the bigger the probability that $Z(n, s)$ yields one, i.e., that an outgoing transition equals zero. We proposed two models **5.1** and **5.2** with

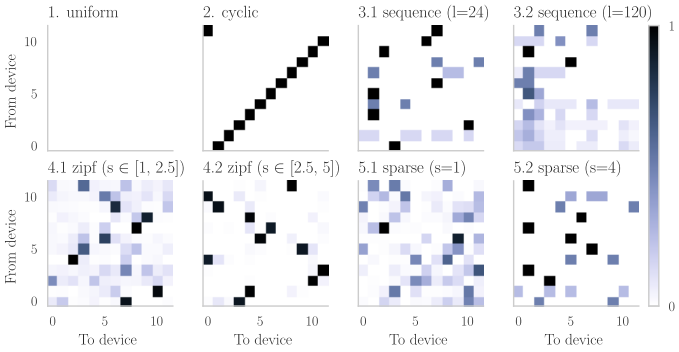

Fig. 5. Heatmaps of the transition matrix $\mathcal{P}_\mathcal{M}$ of each proposed model, with $N = 12$ devices. For each model, the i^{th} row of the heatmap represents the vector of transition probabilities from d_i: $\mathcal{P}_\mathcal{M}(d_i, *)$. Model *1. uniform* has a constant probability of $1/N$.

$s = 1$ and $s = 4$ respectively. We see that $\mathcal{P}_\mathcal{M}$'s sparsity is proportional to the exponent s.

While creating these models, we always ensure that the Markov graph is strongly connected, in order to effectively see the user switch between the N devices, instead of looping through a small subset of $\mathcal{D}$.

To generate sequences of the user's activity for each model, we randomly walk on the Markov graph derived from $\mathcal{P}_\mathcal{M}$, starting from a random device. The resulting sequence S_{tot} is then used to drive the evaluation of the session handoff.

2) Experimental testbed: To evaluate our system, we deploy N virtual devices implementing the SPRINKLER algorithm. We perform one evaluation per behavioral model presented above. From each of them, we obtain an interaction sequence $S_{\text{tot}} = \{r_1, \ldots, r_L\}$ of size L. It is split in two: the first subsequence $S_{\text{init}} = \{r_1, \ldots, r_{L_{\text{init}}}\}$ of size $L_{\text{init}} < L$ is fed to the devices on bootstrap (cf. Section II-B3) to let them know their respective addresses, and to give appliances an initial knowledge of the user's behavior. The second subsequence $S_{\text{exp}} = \{r_{L_{\text{init}}+1}, \ldots, r_L\}$ (of size $L_{\text{exp}} = L - L_{\text{init}}$) provides the interactions performed during the experiment.

In our experiments, a user interaction is atomic: opening and closing a device is instantaneous, and generates a new session. While the SPRINKLER Gossiper algorithm has been effectively implemented by the devices, the Session Handoff is only simulated: based on the (genuine) current device's local sequence, we determine the amount of session data it sends to its peers, and finally compute the network cost C_N and the waiting time T_W for this interaction (cf. Section II-C1).

Experimental parameters: We set the number of devices to $N = 12$. We argue that this number of devices is already three times above current usage behaviors [20]. The initial sequence length L_{init} is set to 30. We consider that a tech-hungry user, owning and regularly switching among twelve devices, would easily achieve 30 interactions per day. Such a sequence length is big enough to contain most devices', yet small enough to provide only a coarse estimation of the user's real behavior. The second subsequence is set to a length of $L_{\text{exp}} = 70$.

The SPRINKLER protocol has three parameters: the Gossiper's fanout f, the Session Handoff parameter γ that controls the overall amount of session data sent, and α that controls the probability of usage below which we do not send any session data to a device. As already stated, we fix $f = \lceil \log(N) \rceil$, because this value has been proven sufficient for a probabilistic broadcast to reach all of its participants with a very high probability [27]. α has been arbitrarily set to $1/(N-1)$. γ is set to 1 when comparing the different usage behaviors in Figure 6 (thus bounding the network cost C_N to the session's size w_{sess}); it varies from 0 to $N-1$ in Figure 7.

According to [37], a web application session can weigh between 10kB to an unbounded value depending on the state-collection method (e.g. snapshots or event logging), and obviously on the application. Hence, we consider that a session weighs between 10kB and 1MB.

B. Evaluation of SPRINKLER

We first evaluate the SPRINKLER Gossiper. Then, we discuss the performance of the proactive session handoff, and of the reactive fallback.

1) Performance of the SPRINKLER Gossiper: The goal of the Gossiper algorithm (Section II-B) is to successfully propagate a user's overall interaction sequence S_t to all devices. To assess the Gossiper's efficiency, we thus compare the size of the real sequence S_t with the local version of the sequence $S_{d,t}$ maintained by each device at that time.

We aggregate the traces from all our experiments (one per user model), thus leading to 6391 studied local sequences. 577 (9.0%) of them are incomplete. Among incomplete local sequences, the median difference from the real sequence ($|S_t| - |S_{t,d}|$) is 1, while the maximal difference is 7 (one tenth of L_{exp}).

We conclude that our algorithm is able to perfectly propagate the sequence of the user's activity most of the time. When not the case, the drift of the local sequence is controlled: the devices eventually get the missing information from other peers, and end up perfectly knowing the user's behavior again. Overall, we believe the local sequences are generally able to generate a fairly unbiased $\mathbf{p}_k$ for the Session Handoff to share session chunks.

Additionally in terms of network cost, each device receives a median amount of activity-related data of 3.5kB per user interaction, leading to a global of 42kB for $N = 12$ devices for each interaction. Hence, the median amount of the global traffic generated by the SPRINKLER Gossiper is of 2.9MB (42kB$*L_{exp}$) per experiment (using a single behavioral model). Obviously, increasing the number of different devices that the user owns inherently has a direct impact on the traffic.

2) The session handoff: We want to understand the kind of user behavior for which our algorithm is best suited. To do so, we compare the waiting times obtained with the different user models for a fixed γ. Thus, we set Sprinkler's parameter γ to 1, i.e., in total, the currently used device can only proactively share as much as the session's size w_{sess}, distributed among the possible next devices.

Remember that a centralized session handoff solution only functions reactively: the normalized user's waiting time T_W is always one in this situation. However, it always scores a proactive network cost C_N of zero.

Figure 6 shows the boxplots of the normalized waiting times (s.t. $T_W = w_{\text{wait}}/w_{\text{sess}}$) for each user model (see Figure 5). The box edges show the first and last quartile, while the whiskers represent 3/2 of the interquartile range. The models are sorted by increasing lower quartile and median. A waiting time of zero means that the entire session was sent proactively; $T_W = 1$ means that none of it was sent, and that the entire session had to be downloaded reactively when the user opened her device. The dotted line represents the median waiting time of the baseline (Section II-C2). It is constant, because the baseline does not take transition probabilities into account.

As expected, the Session Handoff algorithm performs best with the *2. cyclic* model, where the next used device d_{next} is

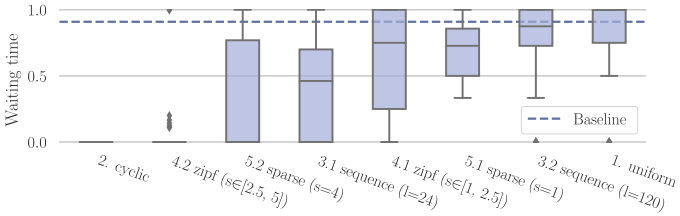

Fig. 6. Box plots of the user's waiting time (normalized) after the session handoff, grouped by behavioral model. Here, $\gamma = 1$: each device shares no more than the session's size to all the other devices. *Lower is better*: a null waiting time means that the session was sent proactively in its totality; a waiting time of one means that the current device needs to download the whole session reactively. We see that the handoff's performance highly depends on the user model.

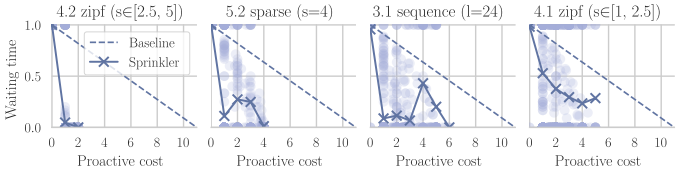

Fig. 7. The user's waiting time (normalized) as a function of the handoff's network cost (normalized by w_{sess}), for several behavioral models, using $\gamma \in [\![0, N-1]\!]$. *Lower is better*: the handoff is more efficient when a small consumption increase yields big waiting time gains.

always known. As a result, the cyclic model scores a constant waiting time $T_W = 0$, meaning that the session is always entirely sent proactively to the right device. The algorithm performs worst with the *1. uniform* model, where d_{next} is unpredictable. It leads to a wait time $T_W > 0.75$ in 75% of the cases. The second best model for the Session Handoff is the *4.2 zipf* with $s \in [2.5, 5]$, where one transition probability greatly overpowers the others in each row (as can be observed in Figure 5). This model is very similar to the cyclic one, with some added noise: the proactive handoff is mostly perfect.

The next two models, *5.2 sparse* ($s = 4$) and *3.1 sequence* ($l = 24$), show fairly similar results. Both models' upper whiskers reach one: the waiting time is highly variable. Figure 5 shows nearly identical transition matrices for these two models: mostly deterministic, apart from some equiprobable transitions. We argue that the difference of median waiting time (0.46 for model *3.1* against 0 for *5.2*) is caused by the uniform probability of switching among 7 devices when using device #1 in model *3.1*. For these two models, we observe that a little loss in predictability of the user's behavior causes more unsteady results, even though the session handoff scores are still good most of the time.

The last three models, *4.1 zipf* ($s \in [1, 2.5]$), *5.1 sparse* ($s = 1$) and *3.2 sequence* ($l = 120$) all show a median waiting time above 0.5 (resp. 0.75, 0.73 and 0.88). The three of them display very dense transition matrices (apart from some constant vectors in *3.2*): we consider them as different types of unpredictable behavior. Indeed, in the *4.1 zipf* model, one transition probability continues to dominate the others, while the two others show many uniform probabilities of transition. This leads to d_{curr} always sending small chunks of the session to several devices, which hampers the overall results.

This experiment shows that the performance of the proactive session handoff greatly depends on the user's behavior. Given our simple inference model, we get better results when the user has predictable habits, despite some variability (e.g. *4.2 zipf*, *5.2 sparse*, *3.1 sequence*).

We performed a second study (Figure 7), this time by varying the γ parameter for a fixed user model, allowing us to observe the influence of the normalized proactive network cost C_N on the waiting time T_W. The figure shows the normalized waiting time $T_W = w_{\text{wait}}/w_{\text{sess}}$ as a function of the network cost C_N, for four different user models, and for every value of $\gamma \in [\![0, N-1]\!]$.

The baseline, represented as a dotted line, shows a linearly decreasing waiting time as the network cost increases. Each dot represents the outcome of a single handoff. The full line represents the average of the waiting time over a sliding window of the network cost. Note that, in our Session Handoff algorithm, the proactive network cost C_N is often lower than γ: indeed, the current device will only send its session to devices that have a non-null probability of being used next according to $\mathbf{p'}_k$ (cf. Section II-C2). In this graphic, lower is better: the *4.2 zipf* ($s \in [2.5, 5]$) model shows a close-to-perfect handoff. Then, we displayed results for *5.2 sparse* ($s = 4$), *3.1 sequence* ($l = 24$) and *4.1 zipf* ($s \in [1, 2.5]$), in the same order that they appear in Figure 6.

Very good session handoff traces will look like *4.2 zipf*: a small increase in the allowed network cost leads to a dramatic decrease in waiting time. Furthermore, *4.2 zipf* is very short tailed: devices never send more than twice the session size. On the other hand, *4.1 zipf* shows poor results: at $C_N = 1$, the average waiting time is above 0.5. The function monotonically decreases until $C_N = 5$: only very unpredictable handoffs cause that much data exchange, thus scoring more than for $C_N = 4$. Finally, *5.2* and *3.1* show a combination of the first and last plots: they attain a very low waiting time at $C_N = 1$ (i.e. deterministic handoffs), and a longer T_W as the network cost increases (i.e., unpredictable handoffs).

We also observe many points at $T_W = 1$: this situation only arises when a device is being used for the first time ever. As it is not in the sequence of interactions, the previous device could not have sent it any session chunks, resulting in a failure of the proactive handoff. Fortunately, once this device joins the gossip, it will be able to proactively receive the session.

Looking at this experiment, we observe two possible use-cases for SPRINKLER: it could either be preferable to keep a bounded network cost, leading to variable waiting times (as in Figure 6); or it could be preferable to have mostly perfect proactive handoffs, while keeping network costs at a reasonable level. Indeed, combining all our models except the uniform one, we see that the proactive network cost C_N never exceeds 7 times the session size, which is far for flooding.

Overall, we find our predictive approach for distributed session handoff promising: for a bounded cost, it can send most of a user's session to her next device in the majority of cases. Future works on the behavioral model (e.g., using timestamps or locations) and on the session handoff decision

algorithm hold further potential to drastically lower the user's waiting time.

3) Reactive handoff: Since our proactive session handoff might be imperfect, we ought to propose a functioning reactive fallback. In a situation where the previous session was only partly downloaded, d_{curr} needs to locate d_{prev} to be able to retrieve the remainder of the last session from it. We assume that d_{prev} is always connected when d_{curr} requests the session.

There are four ways d_{curr} can find d_{prev}'s address:

1) If d_{curr} has an up-to-date local sequence $S_{k,d_{\text{curr}}}$ containing d_{prev} in its last interaction (i.e. d_{prev} is in $S_{k,d_{\text{curr}}}$);
2) If d_{prev} has sent chunks of its session to d_{curr} (that is when $w_{k-1,d_{\text{curr}}} \neq 0$);
3) If the previous and current devices are the same ($d_{\text{curr}} = d_{\text{prev}}$);
4) If d_{curr} wrongly believed that the previous device was d, but d (which knew the right d_{prev}) *redirected* d_{curr} to the right device.

It is only when none of these solutions work that we fail to provide Alice with her last session.

TABLE II
HOW DOES d_{CURR} KNOW THE ADDRESS OF d_{PREV}?

d_{prev} in $S_{k,d_{\text{curr}}}$	$w_{k-1,d_{\text{curr}}} \neq 0$	$d_{\text{curr}} = d_{\text{prev}}$	Redirected	Failed
523 (94.7%)	407 (73.7%)	15 (2.71%)	16 (2.90%)	2 (0.36%)

Table II shows how d_{curr} identifies d_{prev} in all handoff occurrences of our experiment (using the 8 models and $\gamma = 1$). In total, there are $8 * 69 = 552$ data points. Note that the first and second categories are not exclusive. We see that, even though 5% of the used devices have an erroneous sequence during handoff, the current device quasi always finds the address of its ancestor. As a result, the reactive handoff succeeds in 99.64% of the cases.

We still consider a complete failure of the handoff unacceptable. To address this issue, we should avoid relying solely on d_{prev} to retrieve the previous session: indeed, other devices received parts of it. One could for instance accept session chunks from any online device, in a similar fashion to the BitTorrent file exchange protocol [14].

IV. Related work

The problem of providing seamless cross-device session handoff for Web applications has received considerable attention from both the research community and the industry, and nowadays represents a hot topic of research. State-of-the-art works can be divided into different categories according to the different facets of session handoff that they target [19], [34]: (i) triggering the transfer of a session manually or automatically; (ii) dumping and restoring a session; and (iii) forwarding a session either in a centralized or in a distributed manner.

The first category of works is mostly focused on mechanisms for saving/restoring the state of an application in an optimized manner [11], [16], [24], [28], [30], [37], [38]. State saving and restoration are achieved by injecting code into the application's data path, either via code transformation or via proxies that intercept HTTP requests in order to instrument code in the application's Web page. Other facets of session handoff are dealt superficially. The session state is stored on a cloud-based server, from where it is transfered to the target device either by direct transfer, or by providing a URL that the user can load on the target device. Additionally, session handoff is either explicitly triggered by the user, or automatically initiated in a reactive way based on user preferences and context.

A second category of works specifically addresses session migration among a set of servers from a broader perspective (i.e., without considering web applications) [10], [35]. These works rely on SIP signaling protocol to manage the migration. Although it enables forwarding of a session among different servers, it requires a traditional SIP architecture based on SIP servers (i.e., registrar, proxy and redirect server) for session management, making it a centralized solution. Session transfer is either triggered by the user or automatically triggered when the user's IP address changes.

A third category of works addresses session handoff in an *ad hoc* fashion, relying on a peer-to-peer architecture to save and transfer application sessions among devices owned by users [17], [18], [32]. However, these works do not take user behavior into consideration, and as the target device is not known in advance, they rely on blind flooding to propagate the session state.

It is also worth mentioning a number of industry initiatives to provide seamless end-user experience with applications that work across devices, such as Apple Handoff [5], or Google Drive applications. Nevertheless, these approaches are proprietary, centralized and isolated from each other.

Compared to all the aforementioned works, SPRINKLER leverages on the first category, focused on session dumping for Web applications. However, SPRINKLER is based on a totally decentralized approach, and does not rely on the user (or user preferences) to trigger session handoff: it is performed automatically and proactively according to a distributed prediction algorithm. The strength of SPRINKLER is its ability to be decentralized and to predict user behavior in order to optimize session handoff.

V. Conclusion and Future Work

We have presented SPRINKLER, a novel probabilistic dissemination protocol that exploits decentralized learning to enhanced the fluidity of peer-to-peer multi-device user interaction while reducing unnecessary network costs. More generally, SPRINKLER highlights the potential interest of decentralized learning methods to enable private pervasive interactions. In the future, we would like to investigate the use of additional information, such as geolocation and activity duration, to further improve the approach.

Acknowledgments

The authors would like to thank FAPEG and CAPES in Brazil, and CNRS in France for partly funding this work.

REFERENCES

[1] 1 Password. 1password.com.
[2] Adobe PhoneGap. https://phonegap.com.
[3] Amazon Kindle. www.amazon.com/kindle.
[4] Apache Cordova. https://cordova.apache.org.
[5] Apple Handoff. https://developer.apple.com/handoff/.
[6] Evernote. evernote.com.
[7] Google Chrome. www.google.fr/chrome.
[8] Ionic framework. https://ionicframework.com.
[9] Wunderlist. www.wunderlist.com/.
[10] M. Adeyeye and P. Bellavista. Emerging research areas in sip-based converged services for extended web clients. *World Wide Web*, 17(6):1295–1319, Nov 2014.
[11] F. Bellucci, G. Ghiani, F. Paternò, and C. Santoro. Engineering javascript state persistence of web applications migrating across multiple devices. In *3rd ACM SIGCHI Sym. on Eng. Interactive Comp. Sys.*, EICS, 2011.
[12] K. P. Birman, M. Hayden, O. Ozkasap, Z. Xiao, M. Budiu, and Y. Minsky. Bimodal Multicast. *ACM ToCS*, 1998.
[13] P.-Y. P. Chi and Y. Li. Weave: Scripting cross-device wearable interaction. In *ACM CHI*, 2015.
[14] B. Cohen. The bittorrent protocol specification. Technical report, BitTorrent Inc., 2008.
[15] A. Demers, D. Greene, C. Hauser, W. Irish, J. Larson, S. Shenker, H. Sturgis, D. Swinehart, and D. Terry. Epidemic algorithms for replicated database maintenance. *ACM*, 1987.
[16] J. Feng and A. Harwood. Browsercloud: A personal cloud for browser session migration and management. In *WWW '15 Companion*.
[17] E. R. Fisher, S. K. Badam, and N. Elmqvist. Designing peer-to-peer distributed user interfaces: Case studies on building distributed applications. *Int. J. Hum.-Comput. Stud.*, 72(1):100–110, Jan. 2014.
[18] A. Gallidabino and C. Pautasso. Deploying stateful web components on multiple devices with liquid.js for polymer. In *ACM CBSE'2016*, pages 85–90.
[19] A. Gallidabino and C. Pautasso. Maturity model for liquid web architectures. In *ICWE'17*, pages 206–224. Springer.
[20] Google. The new multi-screen world: Understanding cross-platform consumer behavior. Technical report, 2012.
[21] B. Hartmann, M. Beaudouin-Lafon, and W. E. Mackay. Hydrascope: Creating multi-surface meta-applications through view synchronization and input multiplexing. In *PerDis'13*, pages 43–48, 2013.
[22] M. Husmann, N. M. Rossi, and M. C. Norrie. Usage analysis of cross-device web applications. In *PerDis '16*, pages 212–219, 2016.
[23] M. Jelasity, S. Voulgaris, R. Guerraoui, A.-M. Kermarrec, and M. Van Steen. Gossip-based peer sampling. *ACM ToCS*, 25(3):8, 2007.
[24] D. Johansson and K. Andersson. Web-based adaptive application mobility. In *2012 IEEE 1st Int. Conf. on Cloud Networking (CLOUDNET)*, pages 87–94, Nov 2012.
[25] F. Kawsar and A. B. Brush. Home computing unplugged: Why, where and when people use different connected devices at home. In *UBICOMP'13*, pages 627–636.
[26] A.-M. Kermarrec, L. Massoulié, and A. J. Ganesh. Reliable Probabilistic Communication in Large-Scale Information Dissemination Systems. Technical report, 2000.
[27] A.-M. Kermarrec, L. Massoulié, and A. J. Ganesh. Probabilistic reliable dissemination in large-scale systems. *IEEE Transactions on Parallel and Distributed systems*, 14(3):248–258, 2003.
[28] J.-w. Kwon and S.-M. Moon. Web application migration with closure reconstruction. In *WWW'2017*.
[29] L. Lamport. Time, clocks, and the ordering of events in a distributed system. *Communications of the ACM*, 21(7):558–565, 1978.
[30] J. T. K. Lo, E. Wohlstadter, and A. Mesbah. Imagen: Runtime migration of browser sessions for javascript web applications. In *WWW'2013*.
[31] F. Mattern. Virtual time and global states of distributed systems. In M. C. et. al., editor, *Parallel and Distributed Algorithms: proceedings of the International Workshop on Parallel & Distributed Algorithms*, pages 215–226. Elsevier Science Publishers B. V., 1989.
[32] J. Melchior, D. Grolaux, J. Vanderdonckt, and P. Van Roy. A toolkit for peer-to-peer distributed user interfaces: Concepts, implementation, and applications. In *The 1st ACM SIGCHI Symposium on Engineering Interactive Computing Systems*, EICS '09, pages 69–78, 2009.
[33] L. Michal. *Designing Multi-Device Experiences*. O'REILLY, 2014.
[34] T. Mikkonen, K. Systä, and C. Pautasso. Towards liquid web applications. In *ICWE'15*, pages 134–143. Springer, 2015.
[35] W. Munkongpitakkun, S. Kamolphiwong, and S. Sae-Wong. Enhanced web session mobility based on sip. In *The 4th Int. Conf. on Mobile Technology, Applications, and Systems and the 1st International Symposium on Computer Human Interaction in Mobile Technology*, Mobility '07, pages 346–350, 2007.
[36] M. E. J. Newman. Power laws, Pareto distributions and Zipf's law. *Cities*, 30:59–67, feb 2013.
[37] J. Oh, J.-w. Kwon, H. Park, and S.-M. Moon. Migration of web applications with seamless execution. In *VEE '15*, pages 173–185, 2015.
[38] J. Oh and S.-M. Moon. Snapshot-based loading-time acceleration for web applications. In *The 13th Annual IEEE/ACM International Symposium on Code Generation and Optimization*, CGO '15, pages 179–189, 2015.
[39] M. Weiser and J. S. Brown. Beyond calculation. chapter The Coming Age of Calm Technolgy, pages 75–85. 1997.