Dynamic Cooperative Information Display in Mobile Environments

Christophe Jacquet[1,2], Yacine Bellik[2], and Yolaine Bourda[1]

[1] Supélec, Plateau de Moulon, 91192 Gif-sur-Yvette Cedex, France
Christophe.Jacquet@supelec.fr, Yolaine.Bourda@supelec.fr
[2] LIMSI-CNRS, BP 133, 91403 Orsay Cedex, France
Yacine.Bellik@limsi.fr

Abstract. We introduce an interaction scenario in which users of public places can see relevant information items on public displays as they move. Public displays can dynamically collaborate and group with each other so as to minimize information clutter and redundancy. We analyse the usability constraints of this scenario in terms of information layout on the screens. This allows us to introduce a decentralized architecture in which information screens as well as users are modeled by software agents. We then present a simulator that implements this system.

1 Introduction

When people find themselves in unknown environments such as train stations, airports, shopping malls, etc., they often have difficulties obtaining information that they need. Indeed, public information screens show information for everybody: as a result, they are often cluttered by too many items. One given person is usually interested in only one item, so seeking it among a vast quantity of irrelevant items is sometimes long and tiresome.

To improve the situation, we aim at designing an ubiquitous information system that can use multiple output devices to give personalized information to mobile users. This way, information screens placed at random in an airport would provide passengers nearby with information about their flights. To reduce clutter, they would display information relevant to these passengers only.

However, if many people gather in front of a screen, they still will have to seek through a possibly long list of items to find relevant information. One possible solution would be to bring a second screen next to the first one to extend screen real estate. But in the absence of cooperation among the screens, the second one will merely copy the contents of the first one, both screens remaining very cluttered. The solution lies in the judicious *distribution of content* among the screens (see fig. 1).

In this article, we introduce an agent architecture in which neighboring output devices can cooperate to reduce clutter, *without having prior knowledge of each other*. Thus, no manual configuration is ever necessary, and in particular it is possible to move output devices at run time without changing the software

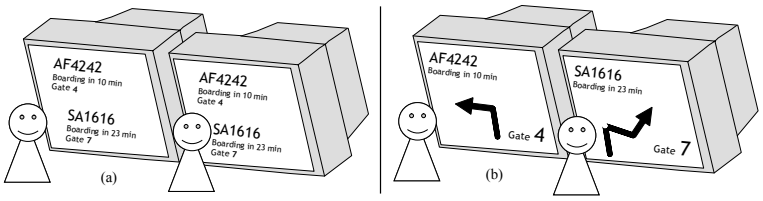

Fig. 1. Two screens, (a) only neighbors, merely duplicating content, and (b) cooperating with each other.

setup. First, we present research work related to these topics. Then, we formally introduce the problem, and draw a list of usability constraints for a cooperative display system. This allows us to propose a solution based on an agent algorithm distributed among output devices and users. The algorithm needs no centralized process: agents cooperate with each other *at a local level* to build a solution. We then present an implementation and a simulator that allow us to test this algorithm. Finally, we introduce perspectives for future work.

2 Related Work

Several systems have been designed to give contextual information to users as they move around. For instance, CoolTown [1] shows web pages to people, depending on their location. Usually, contextual information is given to users through small handheld devices: for example, Cyberguide [2], a tour guide, was based on Apple's Newton personal digital assistant (PDA). Indeed, most experiments in context-aware computing are based on portable devices that give people information about their environment [3].

However, some ubiquitous computing [4] applications no longer require that users carry PDAs: instead, public displays are used, for example the Gossip Wall [5]. In these systems, public displays can play several roles: providing public information when no-one is at proximity, and providing private information when someone engages in explicit interaction, which raises privacy concerns [6].

Though our system uses public displays, it does not provide *personal* information: actually, it provides *public* information *relevant to people located at proximity*. The originality of our system lies in the fact that public displays have not a priori knowledge of their conditions of operation: they can be placed anywhere without prior configuration, giving relevant information to people nearby, and collaborating with each other in order to improve user experience.

3 Problem Statement

For the sake of simplicity, we assume that output devices are screen displays, but this restriction could easily be lifted. We suppose that each user wishes to obtain a given information item called her *semantic unit* (s.u.), for instance her boarding gate. We call *load* of a screen the number of s.u.'s it displays.

We introduce a notion of *proximity* (also called *closeness*). A user is said to be *close to* a screen if he can see its contents. Therefore, this definition includes distance as well as orientation conditions. For instance, if a user turns his back to a monitor while talking in his cell phone, displaying information for her would be totally irrelevant, even if he is at a very short distance of the screen.

In introduction, we have seen that in our system, a user close to (at least) one display must be provided with his s.u. of interest. This is what we call the *completeness constraint*. We have also seen that information must be optimally distributed among screens. To do so, screen load must be minimal so as to reduce clutter (*display surface optimization constraint*). If we consider these two constraints only, the problem would boil down to resolving a distributed constraint system (first criterion) while minimizing the load parameter (second criterion). The problem could thus be seen as an instance of DCOP (Distributed Constraint Optimization Problem); several algorithms exist to solve a DCOP [7].

However, they are designed to find a solution all at once. In contrast, our problem is built step by step from an initial situation. Indeed, we can assume that at the beginning no user is close to a screen. Then, two kinds of events may occur: 1) a user comes close to a screen; 2) a user goes away from a screen. This way, any situation can be constructed by a sequence of events number 1 and number 2. If we assume that we have a suitable solution at one given moment, and if we know how to construct a new suitable solution after an event number 1 or 2 occurs, then we are able to solve the problem at any moment (recursion principle): an incremental algorithm would be highly efficient in this situation.

Optimizing the display surface is an important goal, but it may lead to obtaining unusable systems. Indeed, if a system tries to absolutely avoid clutter, and so always reorganizes screen layout to be optimal, users might end up seeing their s.u.'s leaping from one screen to another every time another user moves. They would then waste their time chasing their information items just to read them, which is worse than having to find an item in a cluttered screen.

Instead, information display should remain pretty much stable upon event occurrence so as not to confuse users. Indeed, if someone does not move, then they may be reading their s.u., so they expect it to remain where it is, not suddenly vanishing to reappear somewhere else. Conversely, when people move, they are generally not reading screens at the same time, so they do not mind if information items are migrated to a new place.

For all this, we take three constraints in consideration.

Constraint C_1 (completeness). When a user is close to a number of displays, his s.u. must be provided by (at least) one of these devices.

Constraint C_2 (stability). A user's s.u. must not move from one display to another, unless the user herself has moved.

Constraint C_3 (display surface optimization). To prevent devices from being overloaded, s.u. duplication must be avoided whenever possible. This means that the sum of device loads must be minimal.

For usability reasons, we consider that C_1 is stronger than C_2, which in turn is stronger than C_3. For instance, let us suppose that three displays show three

s.u.'s for three users (each display shows a different s.u.). Then, if the leftmost user leaves, and a new user arrives on the right, the s.u.'s will *not* be shifted leftwards. This breaks the surface optimization constraint, but we consider it less important than preserving display stability and not disturbing users. However, if at some point the rightmost screen becomes saturated, then s.u.'s will be shifted, so as to ensure completeness, considered to be more important than stability.

4 Solution

In this section, we introduce an algorithm to solve the problem of information display, while satisfying the above constraints. This algorithm is distributed among screens and users, each of them being represented by a software agent.

4.1 Mathematical Formalization

We introduce three costs, the *static cost* of a screen layout, the *dynamic cost* of the action of adding a s.u. to a screen, and the *migration cost* of moving a s.u. from one screen to another.

Let $\tilde{c}$ be a function over $\mathbb{R}^+$, strictly convex and strictly increasing (we will see the reason for this). $\tilde{c}(\ell)$ represents the *static cost* of a screen layout of load ℓ. Thus, for a screen s with load ℓ_s, we define $\mathbf{c}(s)$ to be $\tilde{c}(\ell_s)$. $\mathbf{c}(s)$ is called the *static cost* of the given screen s with its current layout.

Let us suppose that we want to add δ s.u.'s ($\delta \neq 0$) to a given screen s. The *dynamic cost* of the operation, written $\mathbf{d}(s, \delta)$, is defined to be: $\mathbf{d}(s, \delta) = \mathbf{c}(s)_{\text{after operation}} - \mathbf{c}(s)_{\text{before operation}} = \tilde{c}(\ell_s + \delta) - \tilde{c}(\ell_s)$.

Note that the dynamic cost increases as ℓ_s increases, because $\tilde{c}$ is strictly increasing and strictly convex. Thus, δ being given, if $\ell_2 > \ell_1$ then $\tilde{c}(\ell_2 + \delta) - \tilde{c}(\ell_2) > \tilde{c}(\ell_1 + \delta) - \tilde{c}(\ell_1)$. With this definition of a *dynamic cost*, we convey the idea that *the more items are displayed on a screen, the more costly it is to add an incremental item*. Indeed, if there is one item (or even no item) on a screen, adding an item should not increase the time needed to find one's s.u. But if a screen is already overloaded, finding a new item among the multitude will be very long and tiresome.

For instance, let us assume that on a given system, $\tilde{c}$ is defined by $\tilde{c}(x) = x^2$ for two screens, called a and b. Note that this choice for $\tilde{c}$ is totally arbitrary: in practice, $\tilde{c}$ must be chosen for every screen so as to match the screen's proneness to become overloaded. If screen a currently displays 2 s.u.'s, then $\mathbf{c}(a) = 2^2 = 4$; if screen b currently displays 4 s.u.'s, then $\mathbf{c}(b) = 4^2 = 16$. If we want to add one s.u. to these screens, what are the associated dynamic costs? $\mathbf{d}(a, 1) = (2 + 1)^2 - 2^2 = 9 - 4 = 5$; likewise, $\mathbf{d}(b, 1) = (4 + 1)^2 - 4^2 = 25 - 16 = 9$. So if we have the choice, we will then choose to display the s.u. on screen a, which seems to be reasonable. Note that here, both screens share the same static cost function, but in practice each display can define its own static cost function.

We also introduce *migration costs*. A migration cost is taken into account when a s.u. u is moved from one display to another one. Each user U_i interested in the given s.u. contributes a partial migration cost $m(U_i, u)$. The total migration cost is the sum of all partial costs contributed by each user: $m(u) = \sum_i m(U_i, u)$.

4.2 Agent-Based Architecture

Of course, it would have been possible to build a solution around a *centralized* architecture. However, we think that this has a number of shortcomings, namely fragility (if the central server fails, every display fails) and rigidity (one cannot move the displays at will). In contrast, we wish to be able to move displays, bring new ones in case of an event, etc., all this without having to reconfigure anything. Displays have to adapt to the changes themselves, without needing human intervention.

Our implementation is based on software agents that model physical entities: screens are modeled by display agents; users are modeled by user agents. We assume that each agent knows which agents are nearby, and can communicate with them. These assumptions are quite realistic. Proximity of users can for instance be detected by an RFID reader located on a screen, provided that users carry RFID tags sticked to their tickets[3]. As for ubiquitous communications, they are now commonplace thanks to wireless networks.

The agents are *reactive*; they stay in an idle state most of the time, and react to two kinds of events: the appearance or disappearance of an agent at proximity, or the reception of a network message from an agent (which is not necessarily at proximity). In section 4.3, we will see some examples of such messages.

4.3 Algorithm

It is now possible to describe the general behavior of the algorithm. First, note that every user agent references a *main screen*, i.e. a screen where its s.u. is displayed. On startup, all main screens are undefined.

The general layout of the algorithm is as follows: when a user agent either comes close to (*i*) or goes away from (*ii*) a screen, it ponders on doing some operations (described below). So it sends *evaluation requests* to neighboring display agents, to know the costs of these operations. Display agents answer the requests (*iii*, *iv*) and remember the evaluated operations. The user agent has then the choice between either *committing* or *canceling* each of the previously evaluated operations. In practice, the agent commits the operation with the best cost, and cancels all the others.

[*i*] When a user agent with s.u. u comes close to a screen s:

- *if its main screen is already defined*, it sends a `migration-evaluation`(s) request to its main screen. If the result (dynamic cost) of the request is negative the user agent commits it, otherwise it cancels it. This way, a user walking along a row of screens will have her s.u. "follow" her on the screens,
- *if not*, it sends a `display-evaluation`(u) request to s, and systematically commits it (to satisfy constraint C_1, completeness). The s.u. u is sent to the display agent through the network (serialized object).

[3] In this case, only monitors detect the closeness of users. However, the relationship can be made symmetric if a display agent which detects a user agent at proximity systematically sends a notification to it.

[*ii*] When a user agents with s.u. u goes away from its *main screen*, it first sends a `going-away` notification to its main screen, and then:

- *if some other screens are nearby*, it sends a `display-evaluation`(u) request to each of them, and chooses the one with the lowest dynamic cost as its main screen (constraint C_3, display surface optimization). It then sends a commit message to this one, and cancel messages to the others,
- *if not*, its main screen is set as undefined.

[*iii*] When a display agent receives a `display-evaluation`(u) request:

- *if there is still room for s.u. u*, it adds it to its display list: when constraint C_1 (completeness) is satisfiable, the screen tries to satisfy C_2 (stability),
- *otherwise*, it tries to move one of its other s.u.'s to another screen. In practice, for each displayed s.u. v, its sends recursively a `display-evaluation`(v) to each screen seen by every user agent a_i interested in v. The cost of one possible migration (if the corresponding recursive call does not fail), is the cost returned by the call (d), plus the associated migration cost, i.e., $d + \sum_i m(a_i, v)$. If some of the recursive calls do not fail, the display agent chooses the least costly, commits it, and cancels the others. Otherwise, the call itself fails. If C_2 (stability) is not satisfiable, the screen still tries to enforce constraint C_1 (completeness), but while doing so, it still optimizes constraint C_3 (display surface optimization). Rule C_1 is broken only if all neighboring screens have no space left.

[*iv*] When a display agent receives a `migration-evaluation`(s) request to migrate a s.u. u:

- if more than one user agents are interested in u, the call fails,
- otherwise, the display agent sends to s a `display-evaluation`(u) request, and calls the associated cost d_1. It evaluates the "cost" of suppressing u from its display layout. This cost, negative, is called d_2. It calculates the associated migration cost, called m. Then, it returns $d_1 + d_2 + m$. The migration is considered useful if this quantity is negative.

This is the basic behavior of the algorithm. The other operations, such as commits and cancels, are defined in a quite straightforward manner.

5 Implementation and First Results

The algorithm, as well as a graphical simulator (fig. 2) have been implemented. On the figure, users are called `H0`, `H1` and `H2`. Their s.u.'s are respectively `A`, `B` and `C`. There are two screens, called `S0` and `S1`. They can each display at most two s.u.'s. The matrix of 2×3 "boxes" shows proximity relationships: if a user is not close to a screen, the box is empty. If a screen is a user's *main* screen, there is an "M" in the box. If a screen is close to a user, but it is not his main screen, there is a "c" in the box.

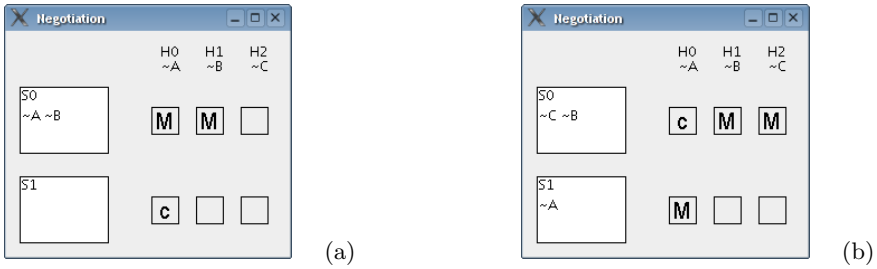

Fig. 2. The simulator introduces a GUI to manipulate proximity relationships.

On figure 2a, `H0` and `H1` are close to screen `S0`, and `S0` is their *main* screen. Thus, `S0` displays s.u.'s `A` and `B`. `H0` is also close to screen `S1`, but it is not his main screen. Then user `H2` comes close to screen `S0`. To satisfy C_1 (completeness), `S0` should display the s.u. `C`, but it can at most display two s.u.'s. So `S0` chooses to break rule C_2 (stability) in favor of C_1, and migrates `A` to screen `S1` (fig. 2b).

The tests performed with this implementation were satisfying. Next, we plan to implement the system in real scale, so as to assess its practical usability.

The system can be used to provide information in an airport or train station, but also for instance to display examination results. In this case, people generally have to find their names in very long lists, which is very tedious. The task would be much easier if only the results of people located at proximity were displayed.

6 Future Work

In this paper, all information output devices were screens, thus favoring the visual modality. However, we are currently finalizing a generalization of the framework presented here to multimodal output devices, handling for instance speech output as well as text output. In this case, users have preferences not only about their s.u.'s, but also about their input modalities. For instance, blind users require information kiosks to provide them with audio information.

Moreover, within a given modality, people can express preferences about the *attributes* of output modalities. For instance, short-sighted people can indicate a minimum size for text to be rendered; people with hearing problems can indicate a minimum sound level for speech output. In short, the attributes are used when *instantiating* [8] semantic units. Note that this extension will have repercussions on cost calculations, since, for example, screen real estate depends on the size used to render s.u.'s textually.

On a given screen, it will be necessary to *sort* the various s.u.'s displayed. This could be done at random, but we think that a *level of priority* could be given to each s.u. This would for instance allow higher-priority s.u.'s (e.g. flights which are about to depart shortly, or information about lost children) to appear first. Similarly, there could be priorities among users (e.g. handicapped people,

premium subscribers would be groups of higher priority). Therefore, s.u.'s priority levels would be altered by users' own priorities.

As seen above, priorities will determine the layout of items on a screen. Moreover, when there are too many s.u.'s so that they cannot fit all on the screens, priorities could help choose which ones are displayed.

In this paper, proximity was *binary*: agents are either close to each other, or away from each other. Actually, it is possible to define several degrees of proximity, or even a measure of distance. These degrees or distances could be used as parameters of the aforementioned instantiation process. For instance, text displayed on a screen could be bigger when people are farther away.

We plan to do real-scale experiments shortly, so the agents in the simulator already rely on Java RMI, so they will be easily deployable on a network. We also plan to test different proximity sensors that can be used to fulfill our needs.

7 Conclusion

In this paper, we have presented a novel mobile interaction scenario: as users are being given personalized information on public displays as they move, displays dynamically cooperate to reduce clutter and increase usability. We have analyzed the diverse constraints of this scenario, which has led us to propose a solution based on a decentralized multi-agent architecture.

This architecture appears to be efficient in simulation. The next step will be a real-scale implementation that will allow field trials with users in context.

References

1. Kindberg, T., Barton, J.: A Web-based nomadic computing system. Computer Networks **35** (2001) 443–456
2. Long, S., Kooper, R., Abowd, G.D., Atkeson, C.G.: Rapid Prototyping of Mobile Context-Aware Applications: The Cyberguide Case Study. In: Mobile Computing and Networking. (1996) 97–107
3. Hull, R., Neaves, P., Bedford-Roberts, J.: Towards Situated Computing. In: ISWC '97, Washington, DC, USA, IEEE Comp. Soc. (1997) 146
4. Weiser, M.: Some computer science issues in ubiquitous computing. Communications of the ACM **36** (1993) 75–84
5. Streitz, N.A., Röcker, C., Prante, T., Stenzel, R., van Alphen, D.: Situated Interaction with Ambient Information: Facilitating Awareness and Communication in Ubiquitous Work Environments. In: HCI International 2003. (2003)
6. Vogel, D., Balakrishnan, R.: Interactive public ambient displays: transitioning from implicit to explicit, public to personal, interaction with multiple users. In: UIST '04, New York, NY, USA, ACM Press (2004) 137–146
7. Mailler, R., Lesser, V.: Solving Distributed Constraint Optimization Problems Using Cooperative Mediation. In: AAMAS '04, IEEE Comp. Soc. (2004) 438–445
8. André, E.: The Generation of Multimedia Presentations. In Dale, R., Moisl, H., Somers, H., eds.: A Handbook of Natural Language Processing. M. Dekker (2000) 305–327