ABSTRACT
One of the central trends in the two last decades was the spreading of mobile systems. While first these devices were intended to be used for stand-alone applications, the trend is emerging towards cooperating mobile systems performing tasks jointly. The aspects of mobility were first considered in the sense of location transparency aiming to hide the problems raised by mobility from the application. However, upcoming applications need to be aware of location and motion requiring simultaneous coordination in space and time. Instead of addressing this problem by programming devices individually, we propose a systemic approach that abstracts from single devices. Two key issues that we address in the paper are the appropriate programming model and scheduling in real space-time. We will show that this can be realized best by the means of a new distributed operating system for mobile systems and present a preliminary design for such an operating system called FlockOS.

Categories and Subject Descriptors
C.2 [Computer Systems Organization]: Communication/Networking and Information Technology

General Terms
Design, Theory

Keywords
real space-time, mobility awareness, distributed operating system

1. INTRODUCTION
The absolute number as well as the relative fraction of mobile systems is increasing steadily. While in the past the number of users and the number of systems were of the same magnitude, in future each user will be equipped with a plethora of mobile devices. Only in rare cases these devices will act in isolation. Instead, they will interact and cooperate with other available devices to provide the desired overall application functionality. For example, devices of different users may cooperatively form ad-hoc networks to provide seamless connectivity to all users. However, in many cases this kind of cooperation will not suffice. In these cases, it may be necessary that several mobile systems perform a task jointly with their activities being coordinated in time and space. This means that the actions taken by a device must be performed at the right time at the right location requiring a real space-time coordination of the devices. For example, robots playing soccer must coordinate their actions in time and space to execute soccer moves successfully. This paper targets at exactly this kind of applications which can be seen as a generalization of classical real-time problems.

Real space-time coordination raises a number of interesting research questions. One central question is which programming abstractions are suitable to program these applications. In our view, a system-oriented view which abstracts from individual devices seems to be most appropriate. This way, we want to relieve the programmer from programming devices individually. Real space-time coordination also requires a shift in the way mobility is dealt with. Usually, it is tried to isolate the problems caused by mobility from the application by establishing location and migration transparency. Contrarily, real space-time coordination requires motion awareness to coordinate the actions of different devices.

A second key issue is the question of how scheduling is
done in real space-time. While for real-time systems established and well researched approaches such as rate monotonic scheduling exist, no scheduling approaches for real space-time are widely accepted and this area can still be considered as a research field with many interesting challenges. For scheduling in real space-time a direct access to all resources of the system and a system-wide scheduling is required. Since these tasks are usually part of the operating system, we argue for a new kind of operating systems that offers support for distributed applications in real space-time. Since the devices have to give up their autonomy in favor of the overall mission, a distributed operating system is required that is more than a collection of local operating systems.

The remainder of the paper is structured as follows: Section 2 defines the problem statement in detail. Section 3 provides general considerations on mobility, motion, and the impact of our approach to device autonomy. Also, some important issues that have to be considered for the design of an operating system aiming to support mobile distributed systems are described. The Sections 4 and 5 are dedicated to the central questions of an appropriate programming model and of scheduling in real space-time. Section 6 introduces the design of FlockOS, our approach to an operating system supporting real space-time coordination. Section 7 discusses related work and Section 8 presents our conclusions.

2. PROBLEM STATEMENT

2.1 Distributed Mobile Systems

From the perspective of the application, the cooperation of systems can be viewed in two ways:

i. **Node view.** Any cooperating system acts separately. Cooperation is done explicitly, e.g., by following certain protocols. Usually, one needs a central coordinator that decomposes the application into subapplications. Each node will execute its subapplication.

ii. **System view (or systemic view).** The cooperating systems form a new system, i.e., cooperating systems are subsystems of the new system. Since an application is the application of the whole (newly formed) system, cooperation is done implicitly.

In this paper, we advocate the second approach where a group of systems constitutes a new system. To distinguish the single systems and the system of systems, we call first ones components or subsystems and the latter meta-system. The concept of meta-systems is far from being new to distributed systems. It provides a number of advantages, especially helpful abstractions to deal with issues introduced by distributed systems. These abstractions are associated with transparency concepts, namely location transparency, migration transparency, error transparency, etc. (cf., e.g., [16]). However, distributed systems consisting of mobile components, performing mobility-dependent tasks require some new considerations since some abstractions do not hold anymore, whereas other abstractions should be kept.

To constitute a meta-system view and to provide the necessary means of abstraction, an entity is needed that represents the meta-system to the application. This entity is usually a common layer shared by all components, i.e., a middleware or an (distributed) operating system. In our opinion, this layer needs rather an operating system characteristic.

In this paper, we discuss a number of issues that appear in the design of an operating system for mobile distributed systems (i.e., meta-systems) and introduce first design concepts for FlockOS, an operating system that is under development by our research groups.

2.2 Distributed Mobile Applications

To gain a more general understanding for the consequences of the suggested approach, we want to consider several mobility-dependent applications that could benefit from this approach. There are several examples for systems consisting of mobile subsystems:

i. Assume the need of a permanent observation of a local event (e.g., an environmental catastrophe) by several earth satellites, each of them not moving in a geostationary orbit. For attaining a continuous observation, the observation task has to be scheduled to different satellites at different times. Applying a classical approach, a developer would decompose the task into a set of single tasks, each of them running on a single satellite. A single task would instruct the respective satellite to perform a given observation in a certain period of time or in a certain sector of its orbit, to process the observation results and to transmit them to other satellites or to the earth station, following given patterns of communication (protocols). However, the manual realization and the subsequent reintegration of single tasks turns out to be complex and error-prone. Furthermore, there is only a restricted flexibility at runtime since after a breakdown of a satellite alternative procedures have to be prepared first. In this paper, it is alternatively suggested to conceive the set of observation satellites as a single system, whose components being at different positions in space at different times.

ii. Two competing societies, FIRA [5] and RoboCup [14], arrange yearly world championships in robot soccer. There exist different leagues that differ, e.g., in size and number of robot players. Most of the leagues allow communication between the robot players, which may be used to coordinate the team. A team of soccer robots can be seen as one system. Following that approach, programming complex strategies and moves could be easier than targeting each single team member.

iii. Already today, personal electronic support devices as PDA, cellphone, or MP3 player are able to cooperate. One can figure several distributed applications that rely on the relative movement of the components. E.g., in guidance systems for pedestrians it could be of importance which direction the pedestrian is heading to. However, even if a single device provided accurate location information, it would be probably much better to have multiple sensors in a distance that allow stereoscopic calculations.
iv. In [18], the authors introduce the concept of a hovering data cloud (HDC). This cloud consists of data that is bound to a fixed location while the systems that store (or even process) the data are not fixed at all. As an application [18] suggests a car that enters an area of special interest (e.g., a traffic jam or crossroads) will be carrier of a HDC. It may get data from other cars in order to enrich its own information or to process it.

If the car leaves this area, it will not be part of the HDC any longer. In this way, the HDC can keep its (approximate) position whereas no car will be part of the HDC system for good.

Please note, that these applications rely on different levels of mobility and require different degrees of determinism and autonomy. A deeper discussion of this concepts is provided in the next section.

3. GENERAL CONSIDERATIONS

3.1 Mobility and Transparency

There exist several views on mobility. Frequently, mobility refers to location transparency (sometimes called spatial transparency) or migration transparency. The probably most popular application of this kind of mobility are cell phones: The system (phone) has to be mobile in the way that it works independently of the location the user transports the device to. One can see the phone as a component in a distributed system (the overall system of phones, base station, connection networks, etc.), where the single phone component provides its function after migration from one to another point of connection. Location or mobility does not matter for the application. The application goal is not to suffer from the mobility introduced by the user. Here, the mobility constitutes disturbances (one could even speak of errors) that have to be tolerated.

Another kind of mobility is linked to the concept of location awareness: A PDA with a GPS receiver can locate the users position (or technically: the PDA’s position) on a map. If it has access to the information (e.g., via web) it can provide the addresses of restaurants one can reach by walk from the current location. Location awareness is opposite to location transparency – a concept of location and the ability to react to location change are vital for the application. However, a concept of motion, i.e., the actual movement itself, is not necessarily needed by a location aware system.

The third kind of mobility accounts for the motion itself, rather than (only) the location. One could speak of motion awareness. Examples of applications with motion awareness are cruise control systems, navigation systems, etc.

In this paper, we consider the last two concepts of mobility, i.e., we do not want location transparency for the application. All application examples from Section 2.2 need information about motion and/or location. However, since we advocate a systemic approach, we want to hide systems internals if possible, i.e., relations between system components (e.g., communication) should be transparent. This introduces another kind of location transparency: a transparency of location aspects within the system, i.e., location aspects of the subsystems. To distinguish the different kinds of transparency we call the latter one distribution transparency.

3.2 Motion

In applications where mobility occurs in the sense discussed in the previous section, it is worth to consider a characterization of motions. Basically, motion can appear in different ways:

- Motion of (or within) the application;
- motion of (or within) the application-executing system;
- combination of both above.

From a mathematical perspective, there is no difference between these ways, since any case can be transformed to any other by applying an appropriate coordinate system. However, from the perspective of the programmer respectively the application it may make a difference, since not all coordinate systems are equivalent in the physical world.\(^2\) Thus, the terms “operating system for mobility-dependent distributed application” or “operating system for mobile distributed system” are each technically not completely accurate, since they reflect only one aspect of the approach, respectively. But from a practical perspective, both share enough to be viewed as one class.

Also, the trait of the motion may differ between different application cases. A motion may be influenced by the application/system (as in case of Example ii); the motion may be known but the system/application has no impact to it (as in Example i), or the motion is neither known nor can be influenced (as in Example iii). Between these cases, there is a wide range of intermediate levels.

3.3 Autonomy

Mobile systems are usually considered as autonomous systems, but a systemic approach to a system consisting of mobile subsystems would limit the autonomy of the subsystems. This is already the case in common distributed systems: The cooperation of the components restricts the local degrees of freedom.

An operating system approach as we advocate here could limit the subsystems’ autonomy even more. Since it introduces a more fine-granular resource management than usual in distributed systems, the subsystems possibly cease to exist as autonomous systems. However, in turn the meta-system gains autonomy. There are arguments that support this claim:

- The meta-system has a higher degree of freedom due to the greater amount of resources.
- The meta-system allows for a more abstract level of control by the application, or by the user respectively.

Thus, the meta-system constitutes an autonomous system. It may seem to be strange to view a system consisting of systems that may change the affiliation to the meta-system as an autonomous system. However, this kind of autonomy is similar to the autonomy of a state or an enterprise, where citizens or employees may also change their affiliation without invalidating the autonomous nature of the cover entity.

These considerations are not only of theoretical interest. To treat a meta-system as an autonomous system has some practical use: It supports a systemic view and helps the programmer to avoid unnecessary micro-management.\(^3\)

\(^2\)Only acceleration-less systems are interchangeable.
3.4 Challenges

The design of distributed mobile systems following a systemic approach introduces a number of issues:

- **Programming model.** Mobile distributed systems need a suitable programming model and application model. Such model should support both, location/motion awareness and distribution transparency.

- **Real space-time scheduling.** Systems considered here are often real-time systems. All examples from Section 2.2 need real-time behavior. Systems that use motion have to consider space in addition to time. This is especially important for scheduling: Such systems need algorithms that take real space-time into account.

These issues are typically targeted by an operating system. Also, in each system consisting of locally distributed mobile components, certain tasks have to be performed often and repeatedly, what also suggests to tackle this at operating system level. Thus, our goal is to design an operating system to support mobile distributed applications. We call our system FlockOS – Federation of linked objects with common tasks Operating System.

The mentioned issues – programming model for motion awareness and distribution transparency and real-space-time scheduling – make the design of FlockOS special.

We are aware, that beside of programming model and real space-time scheduling, distributed mobility-dependent systems introduce a number of other issues, e.g., correctness reasoning on such systems. These issues are subject of further research and will be discussed elsewhere.

4. PROGRAMMING MODEL

4.1 General Considerations

The programming model for FlockOS is intended to help using full capabilities of a distributed mobile system. Since the system’s nodes should appear like one system to both user and programmer, the system’s distribution has to be kept transparent where not needed explicitly. In contrast, the system’s mobility is an essential part of applications for a distributed mobile system. The programming model has to allow describing mobility explicitly. This holds for the system itself and an (abstract) object the system respectively the application refers to, e.g., a ball in a soccer game or a spatial area the system has to enter. Mobility means that constraints both in space and in time have to be considered.

The programming model has to deal with real space-time, in analogy to a real-time system’s programming model that is extended with the consideration of space.

In a system considering real space-time, an application can be bound to certain points in time, points in space or both together. So, the programming model must allow defining these constraints in a flexible way. For an application that is bound to time-constraints, pure time-related constraints have to be specified. Therefore, a well-known model like the periodic task model can be used that specifies the application’s release time, period and deadline, but other time models are also imaginable. While adhering the cyclic task model, time-related constraints can be given through:

\[
\text{timeconstr} = \left( \begin{array}{c} t_{\text{start}}() \\ \text{period}() \\ t_{\text{deadline}}() \end{array} \right)
\]

For an application that is only bound to certain space-related constraints, those can be given by a point respectively an interval on a (time-independent) fixed space-trajectory that has to be specified by the programmer. It describes points in space respectively in the underlying environment. Around this space-trajectory, a geometrical two- or three-dimensional form can be specified, a so-called closure space (CS). It describes the object’s geometrical form, e.g., if the object represents a spatial area the application should be processed within. An application might be bound to time- and space-related constraints but these might not be correlated. The programming model must allow specifying these constraints independent of each other.

When describing the system’s or an object’s motion, time- and space-related constraints have to be considered together. That can be represented by a (time-dependent) closure space around a space-time trajectory (STT). Closure space and STT can be defined by:

\[
cs = \text{closure}([\text{geometric form}(t)]; [\text{parameters}(t)])
\]

\[
stt = \left( \begin{array}{c} x(t) \\ y(t) \\ z(t) \end{array} \right)
\]

Time implicitly represents a third (in the two-dimensional case) or fourth dimension (in the three-dimensional case). If the functions describing that STT are unique, the STT maps points in time to points in space unequivocally. Time-related constraints mapped to an STT determine unique points in space. The other way around, space-related constraints mapped to an STT do not determine unique points in time in either case. So, when time- and space-related constraints are correlated, space-related ones are assigned to time-related ones. As in the pure space-dependent case, the closure space provides further information about the object respectively it specifies further (time-dependent) space-related constraints.

Depending on the application, several STTs must be specified. E.g., one internal STT describes the system’s motion and another one describes an object’s motion or position. The programmer might specify an STT, e.g., for an object with a well-known STT or a predefined position. It also might be a dynamical one that is determined at runtime or it is influenced by the application. So, the programmer rather specifies an end-STT than the object’s current motion respectively position.

An STT refers to a certain environment, e.g., a certain coordinate system. It must be allowed to map several environments to each other when the system and the object are located in different ones. Therefore, the transformation from one system into another or an initial system has to be specified by the programmer.

These described constraints (called space-time constraints) represent a set of resources that have to be allocated respectively in order to process a certain application from the system’s point of view. From programmer’s point of view, an application can be defined by (the object’s) space-time.
constraints to be met and the intended operation along with its required resources.

All hitherto described constraints are rather rigid and hardly consider the system nodes’ cooperation. Instead of fixed time- and space-related constraints, it is desirable that the programming model allows specifying both lower and upper bounds for these constraints.

An application might be influenced by a kind of achievable quality of its results. This quality of service has to be determined, e.g., by the scheduler. The quality of service’s specification highly depends on the application’s issues. Apparently, it is intimately connected with the description of the nodes’ cooperation. General constraints that might be helpful are the minimum or maximum number of nodes required to (cooperatively) process an application as well as a minimal or maximal distance between these nodes.

The following example applications point out different requirements to the programming model by different applications.

4.2 Satellites

In this first example, the description of an application observing a local event by non-stationary earth satellites is examined. In the following, we refer to the local event as the object and the satellites are referred to as the system.

In this example, the object’s space-time trajectory stt_object is stationary, time-independent and might be given in earth coordinates, e.g.:

\[
\text{stt}_{\text{object}} = \left( 51.504876^\circ, -0.017943^\circ \right);
\]

Next, the closure space has to be specified that represents the area to be observed in detail. In this two-dimensional case, a simple geometric field like a circle with its center in the given point stt_object is sufficient. In pseudo-code the closure space might be given through:

\[
\text{cs} = \text{closure(circle}(r = 100m); \text{normal}(0; 0; 1));
\]

In order to make the space-related constraints unique, the circle’s normal is given as optional parameter. In this case, the normal has to be taken as the circle’s surface corresponds with the earth’s one. It is recommended to specify quality of service parameters which correspond to that normal. For example, a maximum observation distance, observation angle or the seamless changeover of observation by different nodes might be given, e.g.:

\[
\begin{align*}
\text{distance}_{\text{max}}&(700km); \\
\text{angle}_{\text{max}}&(10^\circ);
\end{align*}
\]

A greater observation distance respectively angle probably results in fuzzy pictures. In order to complete the first applications example’s constraint specification, time-related constraints have to be given which are independent from the given space-related ones. They could be used as further quality of service parameters. By specifying a period and an equal deadline, the system is instructed to take at least one photo per period. These parameters should be given as an upper bound. Instructing the system to take at least one photo per ten minutes can be specified through:

\[
\text{timeconstr} = \begin{cases} \\
\text{t}_{\text{start}}(\text{now}) & \text{period}(\leq \text{10min}) \\
\text{deadline}(\leq \text{10min}) & 
\end{cases}
\]

Last, the intrinsic application, taking photos and sending them to earth, has to be specified like for usual applications. All object’s space- and time-related constraints described before have to be met by at least one satellite to allow processing the application. In order to allow checking if a satellite is able to meet the constraints, its (internal) space-time trajectory, which represents the satellite’s motion, has to be determined in a way suitable to the chosen object’s coordinate system. In a more general case, the example might be extended by the third space-dimension and more complicated constraints, e.g., a non-stationary object or an adjustable system STT.

4.3 Robot Soccer

Playing soccer demands a great deal on the specification of a programming model. It is a very complex application because it might require the cooperation of several system nodes. In this example, we refer to the two-dimensional case to simplify matters. The ball can be regarded as an object whose primary space-related constraint that has to be met is the object’s position behind the goal line. It can be determined by a time-independent STT specifying a reference point that marks the goal line’s starting point \((x_0; y_0)\):

\[
\text{primary}_{\text{stt}}_{\text{object}} = \left( x_0, y_0 \right);
\]

A rectangular closure space determines the area behind the goal line \((\text{width} x_{\text{width}}, \text{depth} y_{\text{depth}})\). Its bottom left vertex corresponds with the STT-point. A vector to the bottom right point and the rectangle’s depth are given explicitly in order to specify the rectangle unequivocally:

\[
\begin{align*}
\text{cs} &= \text{closure}(\text{rectangle}(\text{width}(x_{\text{width}}, 0); \text{depth}(y_{\text{depth}}))); \\
\text{primary}_{\text{timeconstr}} &= \begin{cases} \\
\text{t}_{\text{start}}(\text{now}) & \text{period}(\text{t}_{\text{game}}) \\
\text{deadline}(\text{t}_{\text{game}}) & 
\end{cases}
\end{align*}
\]

In contrast to the satellites example, here the object’s current STT is capable of being influenced by the system, e.g., by passing the ball. When performing this action, the object’s current STT is replaced by another one.

Considering a soccer game from algorithmic point of view, it is possible to divide it into several soccer moves. Depending on a certain situation, a certain soccer move is performed. This might require the interaction of various nodes that should be kept away from the programmer by the used programming model. To give an example, the following situation where an opponent blocks the object’s (the ball’s) way is illustrated by Figure 1.

In this situation, the application might decide trying to dribble around the opponent instead of passing the ball backwards, for example. Dribbling around the opponent can be achieved in two steps, e.g., moving the ball first from position 0 to position 1 and finally to position 2, as illustrated by Figure 1. It is possible to let Player 0 dribble the opponent, i.e., let it move the ball to position 1 and 2. Moving the ball first to position 1 and then to position 2 can also be realized by Player 0 performing a double pass with Player 1. Player 0 passes the ball to position 1 and
Player 1 passes it to position 2 where Player 0 (or even another player) waits for the ball. That might be faster than the first possibility but requires two (or more) nodes’ cooperation. From programmer’s point of view, this soccer move might be implemented as illustrated by Figure 2. In that example, the system’s distribution is hidden from the programmer. Internal information about the object’s position, which is required for passing the ball, can be regarded as internal temporary space-time constraint. A system node probably has to change its own position in order to reach the object. This position change might be realized by meeting a temporary space-time constraint. However, its specification is intended to be hidden from the programmer but should internally base on the given representation of time and space in order to allow efficient scheduling. The goal’s coordinates and the internal representation of the object’s motion have to be suitable to the environment’s coordinate system. In this example, the coordinate system might be restricted to the soccer field and only considers the two-dimensional case.

From programmer’s point of view, the application is described by an algorithm that tries meeting the ball’s primary space-time constraints by executing several soccer moves. The programmer should care about which one is executed in which situation. A certain soccer move is determined by several ball motions or its end position. The programming model describes these ball motions while keeping their realization away from the programmer. It has to be investigated up to what extent complex soccer moves, that require several intermediate steps, could be given to the system instead of specifying each ball’s motion separately.

4.4 Hovering Data Clouds

Hovering Data Clouds (HDC) are not bound to a predictable location or a fixed time segment. They can “appear from nowhere”. For example, a traffic jam’s location is hard to predict. Depending on its extent, location and duration, the application might perform different actions.

The system’s nodes (the cars) represent the application’s object. The object is defined by a set of nodes that meet certain constraints like a maximum velocity ($v_{\text{max}}$), a certain direction of motion and a maximum distance ($\text{dist}_{\text{max}}$) between each other. So, the set represents the cars that are caught up in the traffic jam. These parameters might be time-dependent. For example, depending on the time the application already holds up, the parameters might be changed in order to catch more (or less) nodes, according to the application issues. So, that set of nodes might be specified through:

$$\text{nodes} = \text{set}\text{nodes}(v_{\text{max}}[t], \text{dist}_{\text{max}}[t],[\text{parameters}(t)]);$$

The application might depend on the traffic jam’s extent, i.e., the number of set members. The object’s spatial extent might be of interest, too. It allows considering the traffic jam as one object and follows the principle of distribution transparency. Therefore, the set’s spatial extent has to be mapped to an internal closure space. This internal closure space is built by the distribution of the set’s nodes. Further, it allows determining the object’s current STT, i.e., the traffic jam’s location and its motion (e.g., in case of bumper to bumper traffic). For example, the traffic jam’s absolute position is necessary to guide following traffic to the right gateway or to precise traffic news.

The programmer can use these parameters, e.g., the set’s (spatial) extent, to compare them with predefined (space-related) constraints in order to specify the application’s action. While the number of involved nodes is easy to handle, the internal closure space’s direction and its definite form can not be predefined. That makes it difficult to compare it with predefined space-related constraints. Therefore, $\text{cs}_{\text{internal}}(\text{set})$ returns the internal closure space containing a set of nodes. A similar instruction to determine the STT is imaginable, too. Comparing the traffic jam’s length with
a given constraint \((\text{length}_\text{object})\) might be realized through:

\[
(\text{length}(\varepsilon_{\text{internal}}(\text{nodes})) \geq \text{length}_\text{object})?
\]

The object’s (spatial) extent might change with time, i.e., cars enter and leave the traffic jam. So, the application itself might change with time although space-related constraints are defined time-independent. An HDC is not bound to fixed time-related constraints in the sense of a fixed starting and end point. There are two ways to bring in time-related constraints. First, they are considered as separate constraints. For example, depending on the time of day or the time the application already holds up, time-related constraints could be used to cause the application performing another action. Second, the space-related constraints could be specified time-dependent, like done for the set of nodes. For example, although the traffic jam’s extent remains constant, the application might change with ongoing time because the space-related constraints that instruct a certain action are weighted with time. So, the time-related constraints can be mapped to space-related constraints and therefore might influence the application.

The described application is characterized by the nodes’ (the cars’) distribution. Their minimum distance or even the number of nodes within the system might be of interest. In contrast to the examples described beforehand, these parameters might be an essential part of an similar application.

### 4.5 Limitations

The described approach using a space-time trajectory to describe the system’s and object’s motion is limited to the feasibility to process that trajectory efficiently. For example, the motion of a person within a crowd that should be observed is hard to predict and the resulting function would be very complex or it would not be possible to specify such a function.

The closure space might be regarded as two- or three-dimensional time-dependent geometric field such as a circle, square, cube or spherical, with the object’s current space-time trajectory location as reference point. In the face of efficient scheduling algorithms, these geometric fields eventually should be restricted to default ones.

Further, the timing-model has several limitations, too. In scope of the robot soccer example, it is desirable to specify time-related constraints in the sense of "as fast as possible". It has to be investigated, if the cyclic task model is sufficient for the multifaceted applications a distributed mobile system is intended to perform.

### 5. SCHEDULING FOR REAL SPACE-TIME

In the field of real-time systems, the cyclic task model is the prevailing one [11]. It is characterized by periodic release times and deadlines corresponding to the release time of the next instance of the task, which is then called a job. Thus, the relative deadline or maximum response time equals the period of the task. Although this approach is very successful and has been implemented in many real-time operating systems, it is not the most powerful one and, by far, not the only one.

The change-over from real-time to real space-time is not just quantitative, but it is a qualitative one, too. Straightforward attempts map space onto time and vice versa, but are not appropriate in the general case. That is why heuristics and statistical approaches are indispensable for non-trivial situations where a simple mapping onto real-time problems is no longer possible. Examples for that can be found in [6, 9, 19].

#### 5.1 Case Study: Satellite Formations

As a starting point, we look at some satellite scheduling problems. The idea of distributed systems, introduced in the 1970s for computers, can be transferred analogously to satellites, resulting in satellite formations working together in a group. It is possible to distinguish between four types of formations concerning satellites’ positions to each other and to the earth.

1. **Geostationary formations.** Both to earth and to each other, the satellites are fixed.

2. **Trailing formations.** All the satellites share the same orbit. There is just a phase displacement between them.

3. **Cluster formations.** All the satellites form a time-stable grid, but at least two of them being in two different orbits.

4. **Constellation formations.** The relative position of the satellites to each other changes with time.

For giving some examples, we can refer to the geostationary communication formation suggested by Arthur C. Clarke in 1945 [1] for the first type. It consists of three geostationary satellites, each of them located at a height of ca. 35,800 km above the earth surface, with an inclination of zero, i.e., straight above the equator. The formation aims at a world-wide communication via satellites. This can almost be achieved by a regular setting of such three satellites, with a phase difference of 120° between two of them. The coverage is near-global, problems occur only in close-to-polar regions with latitudes above ca. 75°. In this simplest case, there is neither a variation in time nor in space. Thus, scheduling with the purpose of maximum coverage is reduced to an equidistant positioning of satellites. The constraint that they have to be placed above the equator comes from physics.

The second type of trailing formations is best described by the so-called A-train, a constellation of six satellites: *Aqua, Aura, PARASOL, CALIPSO, CloudSat* and *Terra*, launched between 2002 and 2008. The distance between two neighboring ones is some hundreds of kilometers, this is about one minute [2]. Such an arrangement is very useful for taking 3D images. Here, scheduling is reduced to a choice of a distance between two subsequent satellites in their orbit.

Cluster formation become more and more important. An example of this third formation type is the planned *Swarm* project [4] with three satellites, two of them flying in a side-by-side formation. Such a formation enables a high-precision measurement of the geomagnetic field, answering the main purpose of the *Swarm* project.

The fourth case is best shown by the Global Positioning System (GPS) consisting of 24 satellites (six planes with four in each of them). The regular arrangement of the orbits and the phases of the satellites guarantees six of them being within line of sight all the time from almost everywhere on the earth. A minimum requirement is four since there are four coordinates \((x, y, z\) and \(t)\) to be calculated. So, there
is always some redundancy. Later it turned out that the further launching of another seven satellites, breaking the regularity of the pattern, improves accuracy, reliability and maintainability much. So today, there are 31 GPS satellites. Scheduling in such a case is non-trivial and requires advanced methods, especially in the non-regular case.

### 5.2 Restrictions in Space-Time

In space-time, one major restriction is imposed by the maximum velocity objects can move. In the very general case, this maximum is the speed of light, resulting in two halves of a light cone, one for the past and one for the future, known from Einstein’s Special Relativity. In most practical cases, bounds on velocity will be much lower. From the viewpoint of time-geography, Hägerstrand made relevant contributions by introducing so-called space-time prisms [8] in an analogous manner to light cones. In both cases, the dimensionality of space is assumed being two. The reason is that a resulting 3d space-time is much better to imagine than the more general and correct 4d version. A lot of low-level questions concerning the feasibility of schedules, such as range and potential duration of interaction [7], can be answered by using this relatively simple approach. Applying this concept in a mathematical fashion, we can introduce the relationship \( R_v \) (which stands for reachability with maximum velocity \( v \)) between two events in the space-time.

\[
e_1R_v e_2 \iff \sqrt{(x_1 - x_2)^2 + (y_1 - y_2)^2 + (z_1 - z_2)^2} \leq vt_1 - vt_2
\]

Note that in (1), \( v \) serves as a parameter. Furthermore, \( t_2 \geq t_1 \) should hold since we would be dealing with time inversion and negative velocities otherwise. The reachability is defined in an optimistic way, since immediate acceleration or deceleration is impossible in practice. Further, we can see that \( R_v \) is a symmetric relationship since we are assuming a general maximum velocity. The symmetry can be destroyed by taking into account individual differing maxima, of course. This is a second reason for seeing the reachability defined in (1) as an optimistic value. That means that one should take it as a necessary, but not a sufficient criterion for reachability in practice. From a third point of view, the approach is a pessimistic one since it is assuming only one of the two events moving. Reachability can be improved dramatically by making both of the events mobile. We will have a closer look on that in the following example.

### 5.3 A Space-Time Scheduling Example

Let’s assume two persons, say Alice and Bob, are separated from each other by 200 km, both having a time slot of 4 h each. They want to meet each other sharing as much time as possible. Their maximum average velocities should be 150 km/h and 100 km/h. How should they do best, i.e., where should they do the meeting and how long can it be?

First, Alice could passively wait for Bob or vice versa. It turns out that Bob could reach Alice’s place, but he had to return immediately, see the dotted curve. Can we do better? Yes, of course, Alice can drive to Bob that would take 80 min one-way. So, she could stay at Bob’s place for 80 min, see the dashed curve. This can be expressed as \( BM1_{100km/h}MB2 \) and \( AL2_{150km/h}MB1 \). In the optimal case, both Alice and Bob have to drive, resulting in a meeting point just beside the street. It has to be at a point sectioning the street correspondingly to the ratio of the average velocities. This ratio is 3 : 2. Thus, they should meet at a distance of 80 km from Alice’s place which means 80 km from Bob’s place. Both of them need 48 min for reaching that point, resulting in the optimal value of 144 min for the meeting itself, see the solid thick lines. The appropriate relationships are: \( A1R_{150km/h}MM1 \) and \( B1R_{100km/h}MM2 \) for the ways to the meeting place and \( MM2R_{100km/h}MB1 \) and \( MM1R_{150km/h}BA2 \) for the ways back, respectively. Have a look at the time-space diagram in Figure 3. Here, the space-time prisms\(^4\) for Alice and Bob are given. Their intersection delivers the solution the above given problem (maximum duration of interaction and place) graphically.

![Figure 3: Graphical solution of a simple scheduling problem about a meeting between Bob and Alice. Their maximal average velocities of 150 km/h and 100 km/h imply the kind of the two space-time prisms. The suboptimal space-time curves are dotted and dashed, the optimal solution is represented with thick solid lines.](image)

### 6. DESIGN

This section describes the design of an operating system corresponding to the considerations above. We call our system FlockOS. This acronym stands for Federation of linked objects with common tasks Operating System. In difference to many other distributed operating system, each node ceases its autonomy in order to allow a fine-grain control by the overall system. Please note, that FlockOS is in a quite early design phase. Thus, some of the descriptions here are a rather preliminary.

In FlockOS an assembly of nodes that participate in at least one application is called a federation. Please note, that a federation may be dynamic: E.g., in case of a hovering data cloud (cf., Example iv in Section 2.2) only cars that are part of the traffic jam or are near the crossroad participate in the federation. An instance of an application is called an activity. We avoid the terms “thread” or “task” since they are usually closely associated with a single execution stream and local resources. In contrast, a FlockOS activity is constituted by certain parts of the execution streams at the local

\(^4\)Here, since space is only one-dimensional, the prisms have collapsed to rhombi.
subsystems. Currently, FlockOS provides no means to ensure security protection\(^5\) and subsystem authentication. We are aware, that – especially for highly dynamic federations – this is an issue that has to be considered in the future.

FlockOS provides dynamic function and role assignments on the base of movement, position and other parameters. Such parameter can be, e.g., the power consumption. Depending on roles and application, FlockOS supports different communication patterns. In case of active moving nodes, i.e., nodes that can influence their motions, FlockOS supports a number of generic motion patterns.

![Architecture of FlockOS](image.png)

**Figure 4: Architecture of FlockOS.**

Figure 4 shows the general architecture of FlockOS. Similar to other distributed operating systems, FlockOS consists of nano kernels at each system component. In case of FlockOS, these kernel need real-time capabilities. The nano kernels support the execution of local parts of activities. Also, the nano kernel is responsible for the access to local resources, especially devices. A communication layer is situated on top of the nano kernels. It provides mechanism for a inter-node data exchange and is the base of the internal coordination of FlockOS. With help of the communication layer and (if present) local sensors FlockOS provides a space-time service that allows for determination of the (absolute or relative) position and motion of the nodes and other objects. If possible, absolute data are gained with means of GPS or similar systems. The space-time service is not a collection of local services: For each object – node or activity-related object\(^6\) – a consensus on position and motion is performed among all nodes that may be concerned of this object until the next consensus round. The motion information are updated regarding the actual drift assumptions. In case of fast objects, small federations, or wide movement spaces, FlockOS can operate in a special mode where space-time service consensus is always global. The space-time service is implemented as an (pseudo-)activity itself.

The Activity Execution Environment (AEE) can be seen as the (macro) kernel of FlockOS: A FlockOS application is written against the AEE. The AEE is responsible for the space-time scheduling of the activities as well as for the motion scheduling of the local nodes (in case the nodes are able to influence their motion). The AEE supports the distribution of code and activity-specific master data which is usually done during the initialization phase of an activity. An update is performed on base of a speculative lazy consistency policy. At a local node, an activity is represented by a local context. The local context of an activity at a node that currently does not perform any action of that very activity is blocked. Please note, that usually an activity consist of more than one active (non-blocked) context at any point of time. In case of a resource shortage, a blocked context may destroyed and later re-established with help of the code distribution facility.

Beside the kernel call interface, the programmer has access to a library of standard operations, e.g., to support object tracking, general formation patterns, or coordinate transformation. Also, the programmer may bypass the FlockOS services and access local nodes directly.

7. RELATED WORK

There exist operating system or middleware approaches to support distributed systems. Most of them try hard to gain location transparency, i.e., to hide matters of location from the user respective programmer. In locally bounded services (cf. [15]), it is referred to real space, but without any distributed aspects. Programming models of operating systems or OS extensions supporting distributed and partial mobility, such as Amoeba [17], Plan 9 [13] or Emerald [10], too, abstract from real space. The problem here is that all of them are restricted to location transparency in their realization of mobility.

Relatively simple group tasks could be solved using the operating system SwarmOS [12]. This operating system allows for communication between neighboring robots. The important features of scalability (easy and automatic integration of new nodes) and error transparency are supported, too. On the other hand, the approach is too closely related to ant algorithms, which are known to be effective and efficient for solving or at least giving approximate solutions to problems like shortest path, travelling salesman, and related ones. Location awareness and motion awareness, which is based on the former, are missing here.

Third, there are wireless sensor networks (WSNs). TinyOS is the most famous operating system for controlling them. It is event-driven and, thus, allows for appropriate reaction to incoming data already on the OS level. This makes it quite convenient for tasks like monitoring. On the other hand, even the cornerstone of real-time support, which is required for space-time scheduling, is lacking in typical WSN operating systems. Additionally, WSNs try hard to obtain location awareness for their nodes. A minimum set of two nodes needs to have the possibility to obtain absolute coordinates. Based on these absolute positions, other nodes can calculate theirs using their relative positions (combining direction and distance) and the mathematical method of triangulation. Again, this approach abstracts away from time, regards location isolated from time. Thus, location awareness can be attained to a certain degree, but motion awareness and space-time scheduling are still missing.

The probably most interesting approach referring to this work can be found in [9]. There are virtual stationary automata (VSA) presented. They are bound to local sections
determined by GPS. If there are members of a set of mobile nodes in such a sector, they take over the execution of VSAa attached to the sector as a substitute. Thus, location awareness is supported, even motion awareness, what is done by the take-over.

8. CONCLUSIONS
We have introduced a systemic approach to mobility-dependent distributed systems, that abstract from single devices of a distributed system but not from location and motion. We identified and discussed two key issues that have to be solved to support such systemic approach: novel appropriate programming models and the scheduling in real space-time. Also, we described the design of FlockOS, our approach to an operating system that supports mobility-dependent distributed applications. Future work will include a further refinement of FlockOS as well as further alternatives for programming models and approaches for real space-time scheduling. In addition, we are looking for proper methods to proof formal correctness of our system.

9. REFERENCES