On Proving the Stability of Distributed Algorithms: Self-Stabilization vs. Control Theory

Oliver Theel and Felix C. Gärtner
Darmstadt University of Technology
Department of Computer Science
D-64283 Darmstadt, Germany
E-Mail: {theel | felix}@informatik.tu-darmstadt.de

ABSTRACT

A particularly suitable design strategy for constructing a robust distributed computer application is to endow it with a self-stabilization property. Such a property guarantees that the system will always return to a specified set of legal states within bounded time regardless of its initial state. A self-stabilizing application therefore has the potential of recovering from the effects of arbitrary transient failures. However, to actually prove that an application self-stabilizes can be quite tedious with current proof methodologies and is non-trivial. The self-stabilizing property of distributed systems exhibits interesting analogies to stabilizing control circuits used in various engineering domains.

In this paper we show that methodologies from control theory can be used to easily prove the self-stabilization property of distributed algorithms. For this demonstration, we use a particular sample application as a test bed for both, traditional proof techniques of the self-stabilization community and methodologies from control theory. It will turn out that the control theory provides adequate techniques whereas traditional proof techniques from the self-stabilization domain exhibit obvious deficiencies.

INTRODUCTION

In a distributed computing environment, multiple potentially heterogeneous computers are interconnected by a communication infrastructure. Distributed applications, running in such an environment, send messages through the communication network and thereby take advantage of the communication infrastructure in order to exchange information. Unfortunately, since distributed applications generally require the co-operation of many components, they are prone to various failures jeopardizing their successful execution. Examples for failures are site failures, communication failures and network partitions. This observation gave rise to demands for enforcing the robustness of distributed applications. More robust applications are expected to tolerate certain kinds and numbers of failures.

A particularly suitable design strategy for constructing a robust application is to design it as a *self-stabilizing algorithm* [2, 11]. Informally, an algorithm has the self-stabilizing property, if – starting from an illegal state – it is guaranteed to return to a specified set of legal states after a finite period of time. Additionally, the set of legal states must be closed under normal system execution, meaning that the algorithm does not voluntarily switch to any illegal state. The definition of legal and illegal states depends on the particular application. Generally, all legal states are specified (e.g., by a state predicate) and illegal states are defined to be those states which are not legal states. This statement seems to be trivial and unimportant but it embodies the entire potential of self-stabilizing algorithms: if the algorithm is guaranteed to reach a legal state from *any* state, then it is able to tolerate *any* kind of failure that perturbs the system state (i.e., any kind of transient failure). The resulting algorithms are capable of tolerating failure scenarios which were unforeseeable at design time. Unfortunately, proving the self-stabilization property of an algorithm is a complicated task [6]. This is particularly true for close-to-reality message passing models of distributed computing, like the so-called *distributed execution model* [4]. Consequently, the research community is highly engaged in finding more adequate proof techniques.

The self-stabilization property of a distributed algorithm as described above exhibits interesting analogies to stable control circuits used in various engineering domains, like electrical and mechanical engineering. Informally, a control circuit is stable, if after a certain period of time, the controlled system reaches and remains in a pre-defined state [5]. Contrary to the self-stabilization research domain, which is a rather new area of research in computer science, control theory in the engineering domain has a century-old background and offers a well-understood theoretical foundation with powerful criteria for reasoning about the stability of control circuits.

The aim of the paper is to narrow the gap between self-stabilization and control theory by adopting criteria originally used for deciding on the stability of control circuits for proving the self-stabilization of distributed algorithms.

In the next sections, we present a specific application scenario, followed by its modeling in terms of a self-stabilizing distributed algorithm. Additionally, the line of reasoning for obtaining a proof of its self-stabilization property is given. A subsequent section models the application scenario in terms of a control circuit and gives the corresponding proof of stability. Finally, a conclusion is drawn. It will turn out that the adoption of control theory is a viable means for obtaining stability statements for self-stabilizing distributed algorithms, which can be modeled as linear, discrete control circuits, in a very simple manner.

APPLICATION SCENARIO

We assume a distributed environment where client processes (also called *customers*) invoke certain services. The invocation of a service is not free of charge: the customer must pay for it by first acquiring and subsequently presenting a capability to the service provider. Such a capability can be thought of as a one-time capability (like a pre-paid service ticket [10]) or can be used for invoking a service multiple times within a given, fixed time interval (like Kerberos session keys [9]). After the time interval has passed, the capability becomes invalid and the customer must acquire a new one, if needed.

Capabilities (also referred to as *tickets* in the scope of this paper) are created by a trusted authority, located at some node of the network. We call this trusted authority *ticket provider*. The ticket provider distributes a certain amount of tickets to *ticket dealers* located at various nodes of the network. The ticket dealers, on their turn, finally sell tickets to customers. Ticket providers generally do not send single tickets but only *packets* of tickets or fractions thereof to the ticket dealers. All packets consist of the same number of tickets. Upon reception of a ticket packet at a ticket dealer, the ticket dealer unwraps the tickets and does some additional bookkeeping. Tickets from a particular packet do not become available to customers all at once, but – due to the sequential processing of tickets by the ticket dealer – one after the other.

The idea behind storing tickets not only at the ticket provider but at the ticket dealers is

1. to balance the load created by selling tickets to customers throughout the network and
2. to avoid an availability bottleneck with respect to service tickets, in case the ticket provider is temporarily unavailable.

To be more precise, the ticket provider tries to guarantee that at any given time a ticket dealer at node i stores exactly q_i ticket packets. The number of ticket packets which are actually stored at dealer i is denoted as $f(c_i)$ with c_i being some local data structure of the ticket dealer. Unfortunately, due to the sequential processing of tickets by the ticket dealer and due to customers purchasing tickets, $f(c_i) \neq q_i$ holds at some points in time. The problem to be solved is to come up with a distributed algorithm that tolerates the behavior of the ticket dealer and which compensates the ticket drift caused by customers. In the remainder of the paper, we concentrate on the co-operation of the ticket provider with a single ticket dealer. Consequently, the index i of the before-mentioned variables will be subsequently omitted. A generalization to an arbitrary number of ticket dealers is nevertheless straightforward.

Ticket provider

The ticket provider's responsibilities in the scenario are as follows. First, the ticket provider must decide how many ticket packets q should be stored at the ticket dealer. This decision may be made by taking demands of the ticket dealer into account, such as ticket processing capability, number of regular customers, or availability of ticket storage at the ticket dealer's node. Many more factors may influence the number of ticket packets for a particular ticket dealer. Due to the generality of the problem, we do not discuss this issue any further. For the scope of this paper, it suffices that a particular number of ticket packets q for the ticket dealer can be identified. Second, the ticket provider must repeatedly compare the number $f(c)$ of ticket packets available at the ticket dealer with the target number of ticket packets q. Finally, if the difference

$u = q - f(c) \neq 0$ then the ticket provider must "react appropriately" such that u will eventually become 0. In general, the reaction of the ticket provider will be one of two kinds: one possibility is the sending of a certain number of ticket packets to the ticket dealer. This behavior is appropriate if $u > 0$. In case of $u < 0$, the ticket provider orders the ticket dealer to invalidate a certain amount of ticket packets. Figure 1 shows the reaction of the ticket provider, if it recognizes a difference of u at time iT_0 (T_0 is the length of the sampling interval as explained later). If $u > 0$

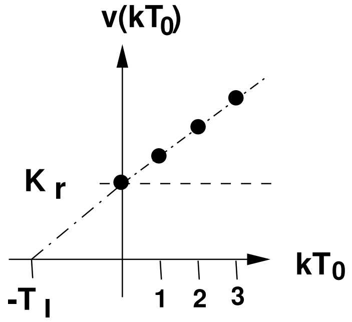

Figure 1: *Behavior of the ticket provider when recognizing a difference of* $u = 1$

then, at time iT_0, the ticket provider sends $u \cdot K_r$ packets, followed by $u \cdot (K_r + (K_r/T_I) \cdot (k - i)T_0)$ ticket packets at time $(k + i)T_0$, $k \in \mathcal{N}$. If, on the contrary, $u < 0$ then the ticket provider notifies the ticket dealer through a message to invalidate $u \cdot K_r$ packets at time kT_0, followed by a notification to invalidate $u \cdot (K_r + (K_r/T_I) \cdot (k - i)T_0))$ ticket packets at time $(k + i)T_0$, $k \in \mathcal{N}$. Note that the precise number of packets sent or invalidated at a particular time t_i depends on the history of differences between q and $f(c)$ detected by the ticket provider at times $jT_0 \leq iT_0$.

Ticket dealer

The ticket dealer's responsibilities are three-fold. First, it must sell tickets to customers. Second, it reports the current number of ticket packets to the ticket provider. This is done through the sending of a special message to the ticket provider. Finally, the ticket dealer receives messages from the ticket dealer. Such a message may either contain new ticket packets or a ticket invalidation order. In the first case, the ticket dealer unwraps the tickets, does some bookkeeping, and finally stores the new tickets until they are destroyed or sold. In the latter case, the ticket dealer invalidates a certain number of tickets according to the invalidation order. If the ticket dealer stores less tickets than required to be invalidated then the ticket dealer will invalidate future tickets upon reception until the required number of tickets has been annihilated. Unwrapping and deletion of tickets cause an identical amount of work for the ticket dealer. Figure 2 shows the behavior of the ticket dealer if a single ticket packet arrives at time t_0. From the figure it can be derived how the avail-

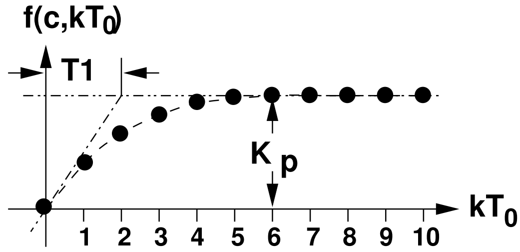

Figure 2: *Behavior of the ticket dealer when one ticket packet is received*

ability of tickets is delayed by the ticket dealer. If a ticket packet is received at time $t_0 = 0T_0$, it is not immediately available for customers. Instead, the tickets will gradually become available as indicated by the graph. Function $f(c, kT_0)$ conforms to some sort of exponential behavior. Eventually, the entire ticket packet has been processed by the ticket dealer and all included tickets become available. Since the ticket dealer cannot produce ticket packets by itself but must receive them from the ticket provider, $K_p = 1$ must hold for the application scenario.

SELF-STABILIZING ALGORITHM

In this section, we analyze the application scenario modeled as a self-stabilizing algorithm.

Model

Apart from constructing an algorithm from scratch, a more efficient way to design and verify a self-stabilizing algorithm for the distributed execution model is:

1. to design and verify the algorithm in the weaker serial model and
2. to transform this algorithm into an algorithm based on the distributed execution model.

The former model assumes communication to take place through communication registers shared by neighboring processes. It additionally assumes that only one process at a time takes an atomic step. The latter model uses message passing techniques and allows an arbitrary subset of processes to proceed simultaneously [8]. This two-step transformation strategy spares us from many intricacies of the distributed message passing model and also makes the algorithm more comparable with the control circuit approach.

Modeling ticket provider and ticket dealer. In the serial model, the application scenario can be modeled using two processes P (ticket provider) and D (ticket dealer) which in effect are state machines. Execution proceeds in synchronous rounds in which both, ticket provider and ticket dealer, take a single atomic step. The ticket provider has access to the value $f(c)$, i.e., the number of tickets stored at the dealer node. If this value differs from the target value q, the provider "sends" ticket packets by updating a local output variable $P.v$ appropriately. (In the notation used, communication variables are prefixed by the name of the node they belong to.) The dealer "receives" the ticket packets by reading $P.v$ and updates its local store $D.c$, which in turn can be read by the provider in the next round.

Figure 3 shows the actual algorithm of the ticket provider using the notation of *guarded commands* [3]. The algorithm consists of a set of actions and each action consists of two parts (which are visually separated by an arrow): the left part is called the *guard* and is a boolean expression over the local state (including the communication variables of neighboring nodes). The right part is an assignment to local variables. Execution proceeds by cyclically evaluating all guards and – from those that evaluate to "true" – nondeterministically executing the corresponding assignment. Thus, if the provider observes that $q \neq f(D.c)$, it sends ticket packets by updating $P.v$ and memorizes the amount of "work" it must do in future cycles by adding the difference to $P.s$. However, if $q = f(D.c)$ it must merely send those ticket packets that are left over from previous send commands and updates $P.v$ according to $P.s$. The algorithm of the ticket dealer is shown in Figure 4. It contains only a single action which is executed in every cycle of the algorithm. It updates the local ticket store c according to the previous value of c (stored in p) and the amount of ticket packets sent by the provider through $P.v$. The consumption of tickets is not explicitly modeled: customers willing to buy tickets must apply for them by different means. Selling tickets is assumed to result in an atomic but

process *Provider P*
 const $P.q$ {* target number *}
 var $P.v$ **init** 0 {* output variable *}
  $P.s$ **init** 0 {* memory *}
 begin

$$\begin{array}{lll} & P.q = f(D.c) & \rightarrow & P.v := K_r \cdot \frac{P.s}{T_I} \\ [] & P.q \neq f(D.c) & \rightarrow & P.v := K_r(P.q - f(D.c) + \frac{P.s}{T_I}); \\ & & & P.s := P.s + P.q - f(D.c) \end{array}$$

 end

Figure 3: *Local algorithm of the ticket provider*

asynchronous modification of c.

process *Dealer D*
 var $D.c$ **init** 0 {* ticket store *}
  $D.p$ **init** 0 {* previous value of $D.c$ *}
 begin

$$\begin{array}{lll} \textbf{true} & \rightarrow & D.c := \frac{K_p \cdot P.v + T_1 \cdot D.p}{T_1 + 1}; \\ & & D.p := D.c \end{array}$$

 end

Figure 4: *Local algorithm of the ticket dealer*

Transformation to a single process. To achieve a correct behavior, both processes must proceed their execution synchronously, meaning that a cycle of both processes must start at the same time and they must use values of the communication registers $D.c$ and $P.v$ that were valid at the beginning of the cycle. This can be modeled by transforming the two algorithms into a single process body. The resulting algorithm is depicted in Figure 5. For sake of clarity, the names of the variables have been left unchanged and the algorithm has been rearranged to stress the linear execution order (actions prefixed with "always" are always executed in one cycle in sequential program order). Now, a single cycle through the (composed) algorithm is equivalent to the execution of one time step in the application scenario. Note that the two assignments from the body of D now bracket the actions of P in such a way, that no buffering of values is needed.

Stable states. To prove that the algorithm self-stabilizes, it is necessary to define the set of legal (i.e., stable) states to which it shall always return. This is usually done by specifying a predicate L on system states. In the application scenario, stable states should obviously be ones where

```
process Combined Algorithm
 const P.q {* target number *}
 var P.v init 0 {* output variable of P *}
  P.s init 0 {* memory of P *}
  D.c init 0 {* ticket store of D *}
  D.p init 0 {* previous value of D.c *}
 begin
        always       →  D.c := (K_p·P.v + T_1·D.p)/(T_1+1)
        P.q = f(D.c) →  P.v := (K_r·P.s)/T_I
   ▯    P.q ≠ f(D.c) →  P.v := K_r(P.q − f(D.c) + P.s/T_I)
        always       →  P.s := P.s + P.q − f(D.c)
        always       →  D.p := D.c
 end
```

Figure 5: *Both algorithms in a single process body*

$f(D.c) = f(D.p) = P.q$. But this is not a sufficient stability condition, because certain values of $P.v$ and $P.s$ can cause a change of the dealer's values in the future. So what values must these two variables have to result in a stable system?

From the algorithm we have a correlation of the values of $P.v$, $D.c$ and $D.p$ expressed in the first assignment:

$$D.c := \frac{K_p \cdot P.v + T_1 \cdot D.p}{T_1 + 1} \tag{1}$$

In stable states $f(D.c) = f(D.p) = P.q$ does not change and so we can conclude the equality

$$f^{-1}(P.q) = \frac{K_p \cdot P.v + T_1 \cdot f^{-1}(P.q)}{T_1 + 1} \tag{2}$$

with $f^{-1}(f(P.q)) = P.q$ which leads to

$$P.v = \frac{f^{-1}(P.q)}{K_p} \tag{3}$$

Having derived a stability value for $P.v$ we can calculate a similar condition for the "memory" $P.s$. From the algorithm and the fact that $P.q = f(D.c)$, only one of the two alternatives is executed in stable states, namely the assignment

$$P.v := \frac{K_r \cdot P.s}{T_I} \tag{4}$$

which, like above, must result in no change of $P.v$ in stable states. With Formula (3) we have

$$\frac{f^{-1}(P.q)}{K_p} = \frac{K_r \cdot P.s}{T_I} \tag{5}$$

and subsequently

$$P.s = \frac{f^{-1}(P.q) \cdot T_I}{K_p \cdot K_r} \tag{6}$$

Now, we can specify the legal states using the following predicate:

$$\begin{aligned} L \equiv\ & f(D.c) = f(D.p) = P.q \wedge \\ & P.v = \frac{f^{-1}(P.q)}{K_p} \wedge \\ & P.s = \frac{f^{-1}(P.q) \cdot T_I}{K_p \cdot K_r} \end{aligned} \tag{7}$$

Proof

In the context of self-stabilization, the stability proof must address two issues [11]:

- (closure) L is closed under normal system execution (i.e., if L holds then L will hold again after one cycle of the algorithm), and
- (convergence) starting from any state, the system eventually reaches a state in L.

Proving closure. The proof of the closure condition must show that the stability predicate L is an invariant of the algorithm. This is straightforward: assume L holds at the beginning of one cycle of the algorithm in Figure 5. The execution of the first assignment changes the value of $D.c$. Using the conditions from L we can calculate this new value to be $(K_p(f^{-1}(P.q)/K_p) + T_1 \cdot f^{-1}(P.q))/(T_1 + 1)$ which results in $f^{-1}(P.q)$.

Because of this fact, the condition $P.q = f(D.c)$ holds and thus the second guarded command is selected for execution. This results in an assignment of $(K_r \cdot P.s)/T_I$ to $P.v$, which – not surprisingly – is equivalent to $f^{-1}(P.q)/K_p$. As $P.q = f(D.c)$, nothing is added to $P.s$ and so its value is left unchanged. Also, the final assignment does not change $D.p$ (because $D.c = f(P.q)$).

So after one cycle of the algorithm, $f(D.c) = f(D.p) = P.q$ holds, $P.v = f^{-1}(P.q)/K_p$ holds and $P.s$ is left unchanged. This means that L holds again, thus L is an invariant of the algorithm.

Proving convergence. The proof of the convergence property of a system is usually achieved by exhibiting a variant function [6] on the system state whose value is bounded from below and which decreases with every step that causes a state change. From such a proof it follows that there will be a point in time where the variant function reaches a minimum and the system will not change its state any-more. These are the desired stable states of the system. However, deriving such a function from a given system requires experience and inspiration since the function must in itself bear the "essence of convergence" of the system. Thus, deriving such functions for arbitrary systems has remained an art rather than a craft.

Returning to our application scenario and the algorithm from Figure 5 again, there are two things to note. First, convergence will probably depend on the concrete values of the constants K_p, T_1, T_I and K_r. Second, while the algorithm is quite simple, the dependencies between individual changes of variables are complex. For example, consider the value of $P.v$. Assume that its value initially is α. Then what can be said about the value of $P.v$ after one cycle of the algorithm? Not very much, since its value depends on (among others) the value of $P.s$. However, the value of $P.s$ depends on $D.c$, which in turn depends upon $P.v$ again.

The effect of such dependency structures on the convergence properties of iteration systems (like the algorithm from Figure 5) have been studied by Arora et. al. [1]. They present a formalism called a *dependency graph* which captures the intricacies of iteration systems. Informally, a variable x is said to *depend on* a variable y if a change of y can lead to a subsequent change of x. The *dependency graph* of an iteration system is a directed graph whose nodes correspond to the variables of the system and whose directed edges correspond to the *depends on* relation. Arora et. al. have shown, that the size of the largest cycle in the dependency graph influences the convergence properties of a system. The essence of their findings is that an acyclic dependency graph is a sufficient condition for convergence. Cycles of length 1 (i.e., self loops) can be handled rather easily but for larger cycles a bounded function must be given.

The dependency graph of our algorithm is depicted in Figure 6. It shows the complex relations between

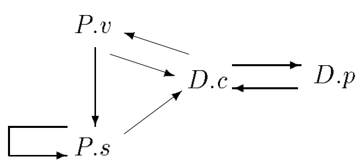

Figure 6: *Dependency graph of the algorithm in Figure 5*

the variables. The fact, that the longest cycle in the graph contains three nodes is a good indication for the difficulty of the problem. Thus, system designers following the self-stabilization approach are let down by its standard proof methodologies. This might be especially disappointing for system designers in this case as simulations strongly support the hypothesis of system stability.

CONTROL CIRCUIT

In this section, we model the application scenario in terms of a discrete, linear control circuit.

Model

Figure 7 shows the modeling of the application scenario as a control circuit for the ticket provider and one ticket dealer. Since the entire control circuit

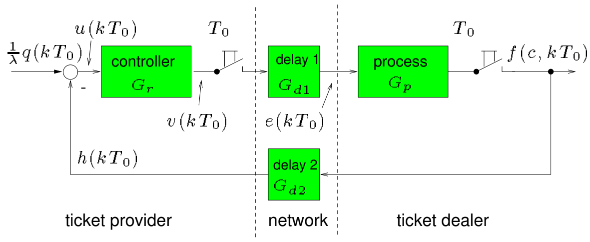

Figure 7: *Application scenario modeled as control circuit*

is implemented based on the components of a distributed computing environment, it is necessarily a *discrete* control circuit. This means that state changes can only occur and are only observed at discrete points $k \cdot T_0$, $k \in \mathcal{N}_0$, $T_0 > 0$, in time. Consequently, $q(kT_0)$ is the target number of ticket packets the ticket provider wants to be stored at the ticket dealer at time kT_0. Analogously, $f(c, kT_0)$ denotes the number of still available ticket packets stored at the ticket dealer at time kT_0. To every discrete point in time kT_0, the ticket provider sends a message containing the newly issued tickets to the ticket dealer. Such a message may be delayed due to a network latency d_1T_0, $d_1 \in \mathcal{N}_0$, i.e., a message sent at time k_1T_0 is received by the ticket dealer at a time $(k_1 + d_1)T_0$. An analogous network latency d_2T_0, $d_2 \in \mathcal{N}_0$, may occur for messages sent from the ticket dealer to the ticket provider. In such a message sent at time k_2T_0, the ticket provider, after reception, is informed about the number of tickets available at the ticket dealer at time $(k_2 - d_2)T_0$. $1/\lambda$ is a correction term which compensates the amplification factor λ of the control circuit. It assures, that – if stable – $q = f(c)$, as given in the description of the application scenario, holds. $u(kT_0)$ is the difference $\frac{1}{\lambda}q(kT_0) - f(c, kT_0)$.

G_r, G_p, G_{d1}, and G_{d2} specify the behavior of the ticket provider, the ticket dealer and the communication network. Their precise specifications will subsequently be given.

Proof

The stability proof for the control circuit is obtained as follows [5]:

(Step 1) the Z transforms of all components of the control circuit are calculated

(Step 2) based on the Z transforms of Step 1, the Z transform of the output-to-input behavior of the entire control circuit $f(c, kT_0)/(\frac{1}{\lambda}q(kT_0))$ is derived

(Step 3) to conclude the proof, it must simply be shown that all singularities of the formula obtained in Step 2 lie within a circle of radius one around the origin in the Z plane

Step 1. Assuming a system with a continuous flow of time, the behavior of the ticket provider (or *controller*) as shown in Figure 1 can be modeled as a linear differential equation

$$v(t) = K_r \left(\sigma(t) + \frac{1}{T_I} \int_0^t \sigma(t)\, dt \right) \tag{8}$$

with

$$\sigma(t) = \begin{cases} 0 & t < 0 \\ 1 & t \geq 0 \end{cases} \tag{9}$$

The Laplace transform of the output-to-input behavior $f(c,t)/v(t)$ is

$$\frac{v(s)}{u(s)} = K_r \left(1 + \frac{1}{T_I s} \right) \tag{10}$$

Then, the Z transform of a corresponding time discrete system with sampling interval T_0 is

$$G_r(z) := \frac{v(z)}{u(z)} = \frac{q_0 + q_1 z^{-1}}{1 - z^{-1}} \tag{11}$$

with

$$\begin{aligned} q_0 &:= K_r & (12) \\ q_1 &:= K_r \left(\frac{T_0}{T_I} - 1 \right) & (13) \end{aligned}$$

The response – modeled as a linear differential equation – of the ticket dealer (or *process*) according to Figure 2 in a continuous-time system to an input $\sigma(t)$ as given by Formula (9) is

$$T_1 f'(c,t) + f(c,t) = K_p \sigma(t) \tag{14}$$

The following formula states the corresponding Laplace transform of the process's output-to-input behavior leading to the Z transform of the process's output-to-input behavior in a time discrete system as given by Formula (16).

$$\frac{f(c,s)}{e(s)} = \frac{K_p}{1 + T_1 s} \tag{15}$$

$$G_p(z) := \frac{f(c,z)}{e(z)} = \frac{b z^{-1}}{1 + a z^{-1}} \tag{16}$$

with

$$\begin{aligned} a &:= -\exp\left(-\frac{T_0}{T_1} \right) & (17) \\ b &:= K_p \left(1 - \exp\left(-\frac{T_0}{T_1} \right) \right) & (18) \end{aligned}$$

In order to be compliant with the modeling of the application scenario as a self-stabilizing system as presented in the previous section, the network transmission delay is assumed being zero. Consequently, the output-to-input behavior of the communication channels with respect to transmission delay is set as given by Formulae (19) and (20).

$$\begin{aligned} G_{d1}(z) &:= \frac{e(z)}{v(z)} = 1 & (19) \\ G_{d2}(z) &:= \frac{h(z)}{f(c,z)} = 1 & (20) \end{aligned}$$

This concludes Step 1.

Step 2. From Figure 7, it can be derived that

$$f(c,z) = G_r(z) \cdot G_{d1}(z) \cdot G_p(z) \left[\frac{1}{\lambda} q(z) - h(z) \right] \tag{21}$$

and

$$h(z) = G_{d2}(z) f(c,z). \tag{22}$$

Formulae (21) and (22) lead directly to

$$\begin{aligned} G(z) &:= \frac{f(c,z)}{\frac{1}{\lambda} q(z)} \\ &= \frac{G_r(z) G_{d1}(z) G_p(z)}{1 + G_r(z) G_{d1}(z) G_p(z) G_{d2}(z)} & (23) \end{aligned}$$

which presents the Z transform of the output-to-input behavior of the entire system. Formula (23) together with Formulae (11)–(13) and (16)–(20) results in

$$G_{app}(z) := \frac{b q_0 z + b q_1}{z^2 + (a - 1 + b q_0) z + (b q_1 - a)} \tag{24}$$

presenting the Z transform of the output-to-input behavior of the application scenario.

Step 3. From Formula (24) is can easily be derived that $G_{app}(z)$ has two singularities at

$$z_{1,2} = -\frac{(bq_0 - 1 + a)}{2} \pm \sqrt{\left(\frac{bq_0 - 1 + a}{2}\right)^2 - (bq_1 - a)} \quad (25)$$

If

$$\sqrt{\Re(z_i)^2 + \Im(z_i)^2} < 1 \qquad \text{for } i = 1, 2 \qquad (26)$$

holds then all singularities lie within radius one around the origin of the Z plane. Thus, the control circuit is guaranteed to be stable.

In the next section, we give an example of a stable control circuit according to the application scenario.

EXAMPLE

For the sample application scenario, we assume a sampling interval of $T_0 = 10$ time units and the following values for specifying the behavior of the ticket provider as well as the ticket dealer:

$$a = -\frac{1}{3}, \qquad b = \frac{2}{3}, \qquad q_0 = \frac{1}{2}, \qquad q_1 = -\frac{1}{4}$$

By using Formulae (12)–(13) and (17)–(18), we obtain

$$T_1 = 9.10, \qquad K_r = \frac{1}{2}, \qquad T_I = 20, \qquad K_p = 1$$

leading to

$$G_{app}(z) = \frac{\frac{1}{3}z - \frac{1}{6}}{z^2 - z + \frac{1}{6}}$$

Then, $G_{app}(z)$ has two singularities

$$z_{1,2} = \frac{1}{2} \pm \sqrt{\frac{1}{12}}$$

for which Formula (26) holds. Consequently, stability of the control circuit is guaranteed. Figure 8 shows the placement of z_1 and z_2 in the Z plane. Finally, we obtain the amplification factor λ for the scenario by

$$\lambda := \lim_{z \to 1} G_{app}(z) = 1$$

Figure 9 shows the number of available ticket packets at the ticket provider as time passes (time is given in units of T_0). In the time interval $t \in [0T_0, 300T_0]$, the ticket provider's policy is to let the ticket dealer store five ticket packets. The graph shows that the desired target number of tickets is eventually precisely met.

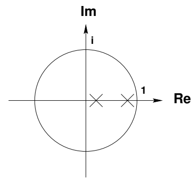

Figure 8: *Singularities of $G_{app}(z)$ in the Z plane*

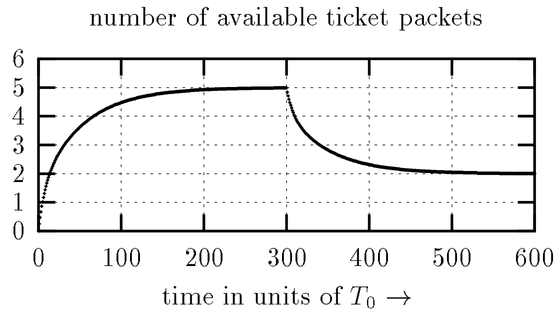

Figure 9: *Asymptotically stable behavior of the sample application scenario*

At time $t = 300T_0$, the ticket provider changes its policy: from now on, only two ticket packets should be available at the ticket dealer. As in the previous case, the number of available tickets is gradually adapted to the desired value. This behavior is not surprising, since the stability of the system has been proven. Therefore, any given target value will eventually be met in the absence of permanent failures. Although not explicitly shown in the figure, a purchase of tickets by customers at the ticket dealer only result in a temporary set back of the number of available tickets. For instance, selling a whole ticket packet at time $t = 150T_0$ would reduce the number of ticket packets from 4.8 to 3.8. In the absence of any further purchases or failures, the system would once again gradually increase the number of available tickets, thereby eventually reaching the target value.

CONCLUSION

Self-stabilizing distributed algorithms are highly needed as the complexity of distributed computing environments steadily increase. More complex distributed computing environments are prone to a larger number of failures which must potentially be compensated by the system if system availability is not to be sacrificed. Self-stabilizing distributed algorithms are promising candidates for designing such environments. Unfortunately, their design and proof of correctness is by no means trivial. In this paper, we showed that alternative approaches for such self-stabilizing algorithms exist. In particular, the modeling as control circuits and their analysis by means of control theory are successful design and proof strategies. For a particular application scenario, we presented a self-stabilizing algorithm and a corresponding control circuit. It turned out, that the proof of correctness in terms of stability can be easily obtained for the latter, whereas the proof for the former poses a tough problem. Additionally, a proof be means of control theory can be constructive, i.e., it indicates how a non-stable control circuit must be modified in order to turn it into a stable one. To the best of the authors' knowledge, there are no practical alternatives within the self-stabilizing research domain to proving the convergence of distributed algorithms via a well-foundedness argument (i.e., a variant function or induction [7]).

Although control theory turned out to be useful for our purposes, there are still unanswered questions. Among them is the following one: Is control theory equally well-suited, if the control circuit modeling a particular algorithm exhibits non-linear properties? Finding an answer to this question will be subject of our future research.

References

[1] Anish Arora, Paul Attie, Michael Evangelist, and Mohamed Gouda. Convergence of iteration systems. *Distributed Computing*, 7(1):43–53, 1993.

[2] Edsger W. Dijkstra. Self stabilizing systems in spite of distributed control. *Communications of the ACM*, 17(11):643–644, 1974.

[3] Edsger W. Dijkstra. Guarded commands, nondeterminacy, and formal derivation of programs. *Communications of the ACM*, 18(8):453–457, August 1975.

[4] Shing-Tsaan Huang, Lih-Chyau Wuu, and Ming-Shin Tsai. Distributed execution model for self-stabilizing systems. In *Proc. of the 14th International Conference on Distributed Computing Systems (ICDCS'94)*, pages 432–439, 1994.

[5] Rolf Isermann. *Digitale Regelsysteme, Band I (in German)*. Springer-Verlag, New York, 1988. ISBN 0-387-16596-7.

[6] J. L. W. Kessels. An exercise in proving self-stabilization with a variant function. *Information Processing Letters*, 29:39–42, 1988.

[7] Y. Lakhnech and M. Siegel. Deductive Verification of Stabilizing Systems. In *Proc. of the 3rd Workshop on Self-Stabilizing Systems (WSS'97), Santa Barbara, CA*, pages 201–216, August 1997.

[8] Masaaki Mizuno and Hirotsugu Kakugawa. A timestamp based transformation of self-stabilizing programs for distributed computing environments. In *Proc. of the 10th International Workshop on Distributed Algorithms (WDAG-10), Springer-Verlag LNCS:1151*, pages 304–321, 1996.

[9] B. Clifford Neuman and Theodore Ts'o. Kerberos: An Authentication Service for Computer Networks. *IEEE Communications Magazine*, pages 33–38, September 1994.

[10] Hans-Henning Pagnia, Christoph Liebig, and Oliver Theel. Mutual Trust between Customers and Providers of Distributed Services. In *Proc. of the International Conference on Networks, Orlando, FL*, pages 87–90. IASTED, January 1996.

[11] Marco Schneider. Self-stabilization. *ACM Computing Surveys*, 25(1):45–67, 1993.